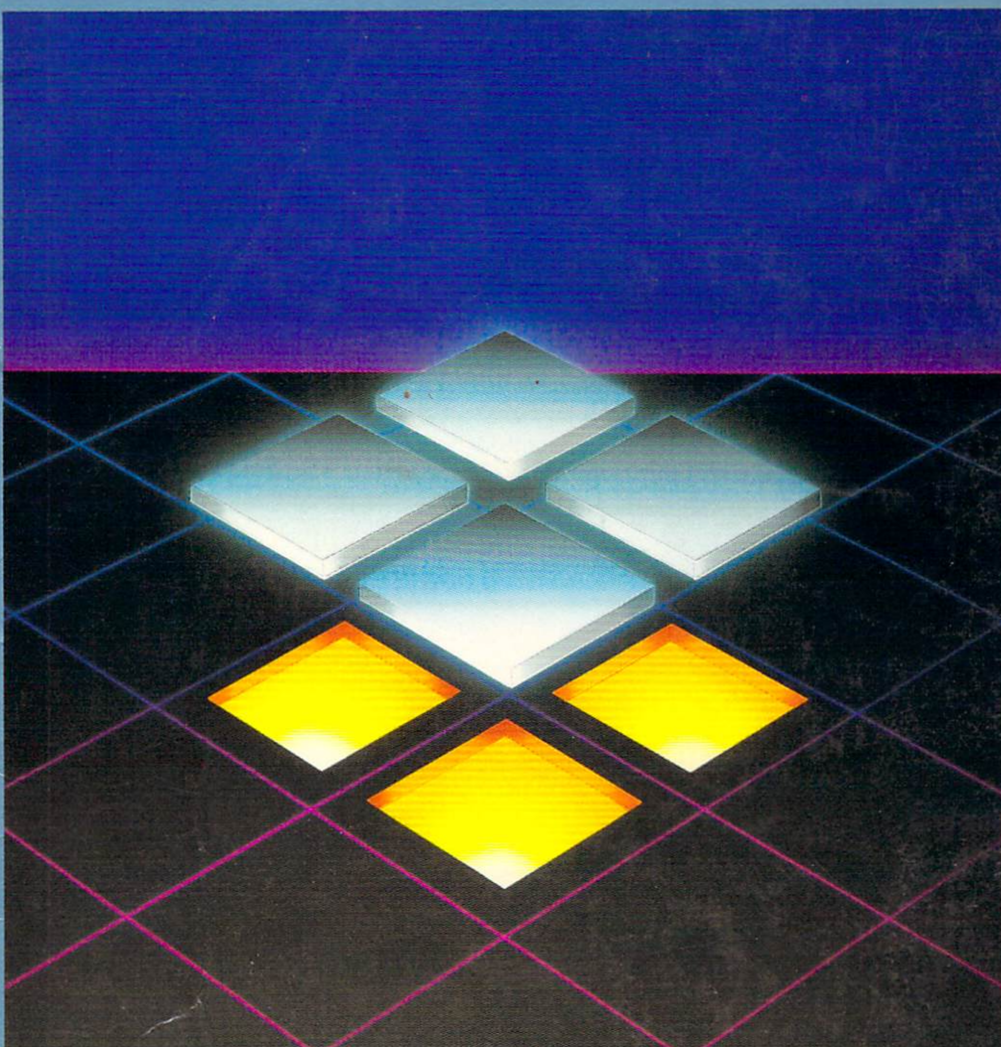
Software Designed
for AMIGA

Lattice C Compiler



Welcome to the Lattice® Family!

You join the many thousands of sophisticated users world-wide using our products. We have all the products you need. And with each Lattice product you get our commitment of *Lattice Service.*

Part of our commitment to you has been our on-going *Lattice Service* policy of timely updates and enhancements. Now *Lattice Service* expands to better serve your support requirements by offering you three ways to access *Lattice Technical Support*:

- On-Line through McGraw-Hill's BIX™ Network.
- On-Line through the Lattice Bulletin Board Service.
- Via the Telephone through The Technical Support Hotline.

The McGraw-Hill BIX™ Network

The BIX (BYTE Information Exchange) Network is a dial-in conference system on which Lattice, in cooperation with McGraw-Hill, has started a SIG (Special Interest Group) for Lattice users.

The BIX Network provides 24-hour availability and world-wide coverage. Once on the network, you have access to Lattice Technical Support as well as a forum of other Lattice users.

In order to use BIX, you need a modem and a VISA or MasterCard. There is a onetime charge of $25 for BYTE subscribers and $39 for those with no BYTE subscription. There is also a connect time charge which ranges from $11 to $14 depending upon usage.

You can subscribe to BIX by calling Tymnet at **(800) 336-0149** and asking for a local access number. Then dial into Tymnet and respond as follows:

a	A single lower case *a* to indicate the baud rate.
byteneti	Entered at the login prompt.
mgh	Entered at the password prompt.
BIX	Entered to indicate that you want the BIX system.
new	Entered once you see the BIX prompt **name?**.

You will then be led through a series of questions (including the charge card to which the session is to be billed). You will be given many chances to change your mind while going through the questions. Only when you complete the set of questions will you be able to get at the Lattice SIG area by entering **JOIN LATTICE.**

Once in the Lattice SIG area you can BIX-mail your questions to the Lattice logname **jriley** for private questions, or you can post your question in the conference mode where Lattice as well as other users can respond. You also have the ability to review other questions and comments posted in the conference mode and check for a new release of your product on our Current Revision List.

If you need additional help on BIX call either BIX-Support at (800) 227-2983 or our Technical Support Hotline.

The Lattice Bulletin Board Service

The **Lattice Bulletin Board Service (LBBS)** is a multi-user bulletin board system for all owners of Lattice products.

The **LBBS** is available 24 hours a day and contains:

- General information on Lattice products.
- Program patches which you can down-load to eliminate bugs located after your product was released.
- Notification of new releases on our Current Revision List.
- Other special information and programs for the serious user.

Using the **LBBS** gives you direct on-line access to Lattice Technical Support without the need to tie up your telephone waiting for the Technical Support Hotline.

You will need a modem to contact the Lattice Bulletin Board. We recommend that you set your communication parameters to from 300 to 2400 baud, 8 data bits, 1 stop bit and no parity.

To reach the **LBBS,** dial **(312) 858-8087**. If you are using 2400 baud, type two spaces to continue after connection. Otherwise, type a return.

A name prompt is the first item displayed. Enter your name and type a return.

First time users will then create a Password before continuing, and then be led through an on-line questionnaire. Please answer all the questions displayed before continuing. You will be asked for this Password at the start of each session on the **LBBS.**

Once you are an **LBBS** user you will be able to leave mail, upload and download files and fill out bug reports. If you have questions on the **LBBS,** please call the Technical Support Hotline.

The Technical Support Hotline

The Technical Support Hotline provides telephone support to all *registered* users of Lattice products.

You can reach the Technical Support Hotline by dialing **(312) 858-0073.**

Support Representatives are available from 9 am to 4 pm (Central Time) Monday through Friday.

It may take a while to get through. Please be patient, we will be with you as soon as we can. An answering machine is used at peak times and at off-hours. You will be called back.

Expect to provide your full name, phone number, product name, product serial number and revision number when you contact the Technical Support Hotline. We will verify that you are a registered user before we can answer your question or return your call.

To make better use of the Technical Support Hotline, please:

- Check your manual first before calling. It may save you the call.
- Have your product serial number and revision number handy.
- Have your specific question ready to insure prompt attention.
- Have an isolated case ready. This should include a sample of the problem you are experiencing.

We cannot take more than five lines of code for testing for specific product problems over the telephone. If a sample is needed in order to resolve your question, either send it to us on diskette or upload it to us using the **Lattice Bulletin Board Service.**

We can only provide support and answer questions relating specifically to Lattice products. We cannot help you write programs while answering technical support questions.

Lattice®
AmigaDOS C Compiler

VERSION 3
PROGRAMMER'S REFERENCE MANUAL

September 12, 1986

Lattice, Incorporated
P.O. Box 3072
Glen Ellyn, IL 60138
TWX 910-291-2190
TELEX 532253
FAX (312) 858-8473

SINGLE COMPUTER LICENSE

The price paid for one copy of the **Lattice AmigaDOS C Compiler** licenses you to install and use the product on only one computer at a time. You may remove it from that one computer and install it on another, but at no time are you allowed to make multiple copies available for others to use. You may install the **Lattice AmigaDOS C Compiler** on file servers in a network, but a separate copy must be purchased for each user. See the diskette envelope for additional licensing terms.

DISCLAIMER

Lattice makes no warranties as to the contents of this software product and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Lattice further reserves the right to make changes to the specifications of the program and contents of the manual without obligation to notify any person or organization of such changes.

ABOUT THIS BOOK......

This is a reference manual for programmers using the **Lattice AmigaDOS C Compiler**. It contains a comprehensive description of the compiler and the library, together with installation and operating instructions. It does not, however, contain complete descriptions of the C language and the AmigaDOS operating system. See the Bibliography in Appendix 1 for a list of books discussing these topics.

The first part of the manual presents general information followed by some supplementary material in the appendices. After that, the bulk of the document consists of the following reference sections:

Section C: Commands
Section D: Data
Section E: Environment
Section F: Functions
Section X: Index

Sections C through M are each preceded by an index listing the symbols defined in that section. Section X is a combined index for all the reference sections.

As with our software products, we've tried to exorcise all bugs from this document before printing. If you spot any errors or omissions, please report them to Lattice Technical Support.

GENERAL INFORMATION

This section presents general information about the Lattice AmigaDOS C Compiler, including installation and operation instructions. It also includes a guide to the lettered reference sections in this manual.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

1. INTRODUCTION

This book is a reference manual for the Lattice AmigaDOS C Compiler. Its primary purpose is to accurately describe all commands, environment variables, header files, and external names that you will normally use while writing and executing C programs. If you are already familiar with C and AmigaDOS, then you should have no problem using this book as a guide to installing and operating the Lattice C Compiler. However, if you are new to C and/or AmigaDOS, you can obtain the necessary background by consulting one or more of the publications listed in Appendix 1, Bibliography.

2. INSTALLATION

The compiler package is distributed on two diskettes which use the Amiga 3.5-inch, double-sided format and contain approximately 880 kilobytes of data.

Appendix 2 lists the contents of the diskettes. One file of particular importance is READ.ME, found in the root directory. This is a text file that contains recent information about the product. Double-click on the "Click For Info On Lattice C" icon or use the AmigaDOS TYPE command to obtain a listing of this file.

2.1 System Configuration

The compiler cannot be used effectively unless your computer is equipped with at least 512 kilobytes of main memory and has one of the following disk configurations:

1. One 3.5-inch floppy drive and one hard drive;

2. One 3.5-inch floppy drive and at least one additional floppy drive, either 3.5-inch or 5.25-inch.

Best results will be obtained with the first configuration, because a hard disk greatly improves performance and simplifies the program development process.

2.2 Installation Procedure

The first step should be the creation of a backup copy of the distribution diskettes. Do this by dragging each distribution diskette's icon onto a blank disk's icon or by using the AmigaDOS **diskcopy** command.

The compiler package is a bootable system. If you are using floppy disks you do not need to perform any special installation procedure. Disk #1 is a bootable disk that leaves you in the CLI environment with the necessary logical names already assigned. When the system prompts you for the Workbench disk during the boot procedure (either from a power-on or ctrl-amiga-amiga sequence), simply insert the compiler disk #1.

Workbench (the icon based environment) is not loaded by this boot procedure. This allows the maximum amount of memory for program development. If you want Workbench, simply type the **loadwb** command at the CLI prompt.

If you wish to set up your own floppy disk based program development environment, Appendix 2 contains a list of the files that are part of the compiler package. You can locate the files anywhere you wish (on any disk), as long as you make the logical name assignments described in Section E.

Physical installation is only necessary when a hard disk system is going to be used. The recommended hard disk installation procedure is to execute the **install_hd**

batch file in the S directory on disk #1. Several subdirectories will be created during this process. These subdirectories contain the header files and a number of C and assembler source files. For example,

```
execute Lattice_C:s/install_hd
```

creates the directory **SYS:lc** if necessary. Then it copies the executable files to the **C:** directory, the header files to the directory **SYS:lc/include,** the libraries and object files to the directory **SYS:lc/lib,** and the run-time source files to the directory **SYS:lc/source.** It will not copy the examples from the floppy disk.

Some of the source files included are for demonstration purposes, while others contain parts of the Lattice run-time system that you may desire to change.

2.3 Environment Variables

For proper use of the compiler, you should make several logical name assignments and add the directory containing the compiler executable files to the **PATH** environment variable. The assignments which should be made are **INCLUDE:, LC:, QUAD:,** and **LIB:.**

The **PATH** variable is used by AmigaDOS version 1.2 or later when it is loading programs for execution. If the specified program is not in the current directory, then AmigaDOS searches each of the directories mentioned in **PATH.** For example, if programs can be found in "DHØ:c" and "DHØ:lc", then you should use the following command to establish the **PATH** variable:

```
path DH0:lc add
```

The **INCLUDE:** assignment is used by the compiler to

locate "header files" that are fetched via the **#include** operation. It should refer to the directory in which you normally keep header files. For example,

```
assign INCLUDE: DH0:lc/include
```

or

```
assign INCLUDE: "Lattice C:include"
```

The **LC:** assignment allows the driver program to find the two compiler phases if they are not in the current directory or the **C:** directory. For example,

```
assign LC: DH0:c
```

or

```
assign LC: "Lattice_C:c"
```

The **LIB:** assignment is used to identify the location of the run time support files and libraries for linking. If this assignment is not made, the driver program **lc** assumes the libraries reside in the directory **SYS:lc/lib**. For example,

```
assign LIB: "Lattice C:lib"
```

or

```
assign LIB: DH0:lib
```

The driver program uses the **QUAD:** logical name assignment to allow the default location of the compiler intermediate file to be someplace other than the source file directory. Placing the quad file on a different drive or in ram-disk can significantly improve compile times.

Logical name assignments may also be used as a form of shorthand when typing AmigaDOS commands. For instance, it is much easier to type

```
LIB:lc.lib,LIB:amiga.lib
```

than it is to type

```
DH0:lc/libraries/lc.lib,DH0:lc/libraries/amiga.lib
```

when manually invoking the linker.

These assignments should be added to the file **s/startup-sequence** on the Workbench disk.

What assignments you make depend on the installation method you use, as follows:

1. Hard Disk Installation:

   ```
   assign LC: DH0:c
   assign INCLUDE: DH0:lc/include
   assign LIB: DH0:lc/lib
   ```

   This assigns **LC:** to directory **c** on the hard disk **DHØ:**, **INCLUDE:** to subdirectory **SYS:lc/include** and **LIB:** to subdirectory **SYS:lc/lib.**

2. Floppy Disk Installation:

   ```
   assign LC: "Lattice_C:c"
   assign INCLUDE: Lattice_C:include
   assign LIB: Lattice_C:lib
   assign QUAD: RAM:
   ```

   This assigns **LC:** to the **c** directory of the floppy disk volume named **Lattice_C, INCLUDE:** to the directory **include** on the same floppy, and **LIB:** to the subdirectory **lib** of the same floppy disk volume.

Note that these assignments are made automatically if you boot the compiler disk #1.

If you install the compiler in some other way, you should make whatever assignments are appropriate to enable **lc** to find the various files.

3. OPERATION

The Lattice AmigaDOS C Compiler uses the classical edit-compile-link sequence for program development. In other words, you must first create a source file containing the C program. This is fed to the compiler, which produces an object file. Object files then serve as input to the AmigaDOS linker, which produces an executable file.

3.1 Using the Compiler and Linker

Source files are simply normal AmigaDOS ASCII text files, consisting of text lines separated by linefeeds. AmigaDOS includes a text editor named ED that you can use to produce suitable C source files. Also, there are many powerful text editors available from various software vendors, including the **Lattice Screen Editor (LSE)**. For extremely simple programs, you can just copy the source lines from your keyboard to the source file, as shown in the following example:

```
copy * hello.c
#include <stdio.h>
void main() { printf("Hello world!\n"); }
CTRL-\          [Hold CTRL while pressing '\']
lc -L hello
hello
```

These steps assume that you have installed Lattice C in

the normal way and that your PATH and logical name assignments were made as suggested in the previous section.

If everything works OK, then you will see "Hello world!" on your screen immediately after typing the last line. According to the venerable Kernighan and Ritchie book, you have now overcome the first hurdle on the path to learning a new language or compiler system.

Now let's examine what you did. The first line

```
copy * hello.c
```

is an AmigaDOS command that begins copying lines from the console (i.e. your keyboard) to a file named "hello.c". If the file doesn't exist, AmigaDOS creates it; if it exists, AmigaDOS clears it. Each line you type will now be written to the file until you type a line consisting of a CTRL-\ character.

Then the command

```
lc -L hello
```

invokes the two phases of the compiler to compile "hello.c". After this command, you should see some Lattice copyright messages on your screen, but you should not see any error or warning messages. If you do, then there is something wrong with the source program in "hello.c" or else you have not set PATH and/or INCLUDE correctly.

If the program was compiled successfully, **lc** invokes the linker, specifying the input files and library files. Again, you should see some copyright information, but there should be no error or warning messages.

The final command

```
hello
```

executes the program named "hello" that was created by the linker. And of course, this program simply prints "Hello world!" to the screen and then terminates.

Now if you were not able to complete these steps with satisfactory results, you should contact Lattice Technical Support or consult with a friend who has more AmigaDOS and/or Lattice C experience. We've found that first-time users often get hung up at this point and waste a lot of time because of simple "cockpit problems".

3.2 Using the Math Library

When you're comfortable with the "Hello world!" program, try compiling and executing some of the demonstration programs that are normally installed in the "source" directory. For example, the source file "ftoc.c" converts temperatures from Fahrenheit to Celsius and requires that you link with the math library. You can run this program via the following commands:

```
cd LC:source
lc -Lm ftoc
ftoc
```

Again, you must have installed the compiler in the standard way and made the appropriate logical name assignments. Note that we specified including the Lattice math library **lcm.lib**. This is necessary whenever you use the "double" or "float" data types in your program or when you call any of the functions defined in the **math.h** or **float.h** header files.

There are actually three math libraries supplied with the Lattice AmigaDOS C Compiler. The standard math

library **lcm.lib** contains all the math functions documented in this manual and supports the **IEEE** format for representing floating point numbers. The math library **lcmffp.lib** supports the Motorola Fast Floating Point **(FFP)** library supplied by Commodore/Amiga, and contains a subset of the standard math routines. It should be used whenever you compile a program with the fast floating point option **-f**. This format and the routines that are available are documented in the Amiga ROM Kernel Manual. The third math library **lcmieee.lib** provides a linkage to **IEEE** format routines supplied by Commodore/Amiga on the Workbench diskette. It contains low level routines such as add, subtract, multiply, and divide that can be used instead of the same routines in the standard math library. This library can be used to utilize the limited support of the floating point coprocessor that Amiga may provide as part of their **IEEE** floating point support. Note that this library does not contain any transcendental functions. If it is specified, you must also specify the standard math library.

When you manually link a Lattice C program, the standard Lattice run time support library **lc.lib** should always be the last one mentioned before Amiga's run time support library **amiga.lib**. If your program uses floats or doubles, the appropriate math library should be mentioned just before the standard library. So in order to link **ftoc** for the **FFP** format you would use the following linker command line:

```
blink lib:c.o,ftoc.o to ftoc lib lib:lcmffp.lib,lib:lc.lib,lib:amiga.lib
```

3.2.1 Using Other Libraries

There are many useful libraries available for the Lattice AmigaDOS C Compiler, both from Lattice and from

other software publishers. Usually the library documentation will explain how to link with Lattice object modules, and some products may even include linking batch files.

The general rule for using other libraries is to always mention them before the Lattice math and standard libraries. If you use the driver program **lc**, you may specify additional libraries immediately after the **-L** option. See Section C for a complete description of the **lc** command.

3.2.2 Including Other Object Modules

In some situations you may want to include more than one object module when linking. For example, if your application consists of two object modules named **mymain.o** and **myfuncs.o** and you want the executable program to be named **myapp**, then you could invoke the linker with the following command:

```
blink lib:c.o,mymain.o,myfuncs.o to myapp lib
lib:lcmffp.lib,lib:lc.lib,lib:amiga.lib
```

3.2.3 Advanced Features

Once you're comfortable building simple C programs, you will probably want to become familiar with the compiler's advanced features, such as different addressing modes and special compilation options. This information is all presented in Section C, which contains detailed specifications for the compiler and linker commands.

4. LATTICE C LANGUAGE DEFINITION

This section describes the C language as supported by Version 3 of the Lattice compiler, which conforms closely to the compilers associated with UNIX System V. Fortunately, the ANSI committee for C standardization has chosen to use the UNIX V definition as their starting point, and so the Lattice compiler also conforms to most of the proposed ANSI standard.

Of course, the committee has also proposed several additions to the language, and Lattice is actively participating in this refinement process as a committee member. The most important of these additions, a feature called "argument type checking", has been incorporated into the Lattice compiler. The others will be added as they become better defined.

Notice that we are not providing a complete C language specification in this reference manual. Appendix 1 cites several books that do this very well, including the Kernighan and Ritchie (K&R) text which began the popular movement towards C. We recommend that you obtain K&R plus one of the more recent C programming books that discuss the latest UNIX and ANSI language features.

In this section, we'll assume that you have a general knowledge of the language obtained from such books and/or from actual experience with another C compiler. Then we'll describe the peculiarities of the Lattice compiler, which fall into three categories:

1. How does the C language as supported by Lattice differ from the specification provided in K&R? We've chosen to use K&R as our basis for comparison simply because the document describing ANSI's proposed C standard is not yet readily available.

2. How does Lattice C handle the murky areas of the language that are usually written off as "implementation issues"?

3. What are the processing limits of the Lattice compiler?

Also, if you have been using an earlier version of Lattice C, you will find Version 3 to be considerably different, both in language features and in library features.

4.1 Comparison to K&R

The most precise definition of the C programming language generally available is in Appendix A of the Kernighan and Ritchie text, which is entitled **C Reference Manual.** The following list highlights the differences between Lattice C and the K&R definition. The items use the K&R numbering scheme. For example, **CRM 2.1 Comments** refers to paragraph 2.1 in the K&R C Reference Manual.

CRM 2.1 Comments

Although the default mode is that comments do not nest, a compile time option can be used to allow nested comments. Here's an example of nested comments:

```
/*  Begin comment block #1
 .
 .
/*  Begin comment block #2 (nested within block #1)
 .
 .
*/  End comment block #2
 .
 .
*/  End comment block #1
```

CRM 2.3 Keywords

Lattice C includes two additional keywords, **void** and **enum**. These are ANSI-compatible.

CRM 2.4.1 Integer constants

Names declared as values for an enumeration type may be used as integer constants.

CRM 2.4.3 Character constants

Two new escape sequences are recognized:

\v Specifies a vertical tab (VT) character.

\x Introduces one or two hexadecimal digits which define the value of a single character. For example, '\xf9' generates a character with the value 0xF9.

Although by default the compiler permits only a single character to be defined, a compile time option can be used to permit multiple character constants. The result will be a short integer for a two-character constant and a long integer for three or four characters.

CRM 2.5 Strings

The same \x convention described above can be employed in strings, where it is generally more useful. In addition, a compile time option can be used to force the compiler to recognize identically written string constants and only generate one copy of the string. Note that a quoted string used to initialize a character array is not actually treated as a string constant, since it is actually placed into the array at compile time. For example,

```
char abc[] = "1234";
char *p = "1234";
```

produces a 5-byte array named **abc** containing the digit characters from 1 through 4, followed by a null byte. The second declaration produces a string constant similar to the contents of **abc** and places a pointer to the constant into **p**.

CRM 2.6 Hardware characteristics

See Section 5 of this reference manual.

CRM 4. What's in a name?

Each enumeration is conceptually a separate type with its own set of named values. The properties of an **enum** type are identical to those of **int** type.

The **void** type is used when declaring a function that has no return value.

CRM 6. Conversions

An expression can be converted to the **void** type by means of a cast. This is often used to explicitly indicate the discarding of a function return value. An expresion of type **void,** however, cannot itself be converted or used in any way.

CRM 7.1 Primary expressions

The Lattice compiler always enforces the rules for the use of structures and unions so that it can determine which set of member names is intended. Since the compiler maintains a separate set of member names for each structure or union, the primary expression preceding a . or -> operator must be immediately recognizable as a structure or as a pointer to a structure of the type that contains the specified member name. For example, given these declarations:

```
struct A
  {
  int x;
  int y;
  } m,*p;

struct B
  {
  int xx;
  int yy;
  } n,*q;
```

the following statements would be invalid:

```
p = &n;    /* p must point to structure type A */
q = &m;    /* q must point to structure type B */
```

You can, of course, convince the compiler that you really want to generate code for these statements by using the appropriate cast operations, as follows:

```
p = (struct A *)(&n);
q = (struct B *)(&m);
```

CRM 7.2 Unary operators

The requirement that the & operator can only be applied to an lvalue is relaxed slightly to allow application to an array name (which is not considered an lvalue). Note that the meaning of such a construct is a pointer to the array itself, which is quite different from a pointer to the first element of the array.

The difference between a pointer to an array and to an array's first element is only important when the pointer is used in an expression with an integral offset, because the offset must be scaled (multi-

plied) by the size of the object to which the pointer points. When the pointer points to the array instead of to the first array element, the target object size is the size of the whole array, rather than the size of a single element.

CRM 7.7 Equality operators

The only integer to which a pointer may be compared is the integer constant zero.

CRM 7.14 Assignment operators

Both operands of the simple assignment operator (=) may be structures or unions of the same type.

CRM 8.1 Storage class-specifiers

The K&R text states that the storage class-specifier, if omitted from a declaration outside a function, is taken to be **extern.** This is somewhat misleading, if not plainly inaccurate. In fact, as the K&R text points out in CRM 11.2, the presence or absence of **extern** is critical to determining whether an object is being defined or referenced. If **extern** is present, then the declared object either exists in some other file or is defined later in the same file. But if no storage class specifier is present, then the declared object is being defined and will be visible in other files. If the **static** specifier is present, the object is also defined but is not made externally visible.

The only exception to these rules occurs for functions, where it is the presence of a defining statement body that determines whether the function is being defined.

The Lattice compiler can be forced to assume **extern** for all declarations outside a function by means of the **-x** compile time option. Declarations which explicitly specify **static** or **extern** are not af-

fected.

CRM 8.2 Type specifiers

Two new type specifiers are supported, as mentioned earlier: **void** and **enum.** Also, the compiler recognizes the following new types that are specified via multiple keywords:

unsigned short
unsigned short int
unsigned long
unsigned long int

CRM 8.5 Structure and union declarations

The Lattice compiler maintains a separate list of member names for each structure and union. Therefore, a member name may not appear twice in a particular structure or union, but the same name may be used in several different structures or unions within the same scope.

Also, structure and union tags are in the same class, which means that you cannot use the same tag for both a structure and a union.

Enumerations are declared in much the same way as structures and unions, with a separate list of identifiers for each enumeration type. Enumerations are unique types which can only assume values from a list of named constants. The language treats them as **int** values but restricts operations on them to assignment and comparison. The named constants, however, may appear wherever an **int** is legal.

The optional name which may follow the keyword **enum** plays the same role as the structure or union tag; it names a particular enumeration. All such names share the same space as structure and union tags.

The names of enumerators in the same scope must be distinct from each other and from those of ordinary variables.

CRM 8.7 Type names

Although a structure or union may appear in a type name specifier, it must refer to an already known tag, that is, structure definitions cannot be made inside a type name. Thus, the sequence:

```
(struct { int high, low; } *) x
```

is not permitted, but

```
struct HL { int high, low; };

(struct HL *) x
```

is acceptable.

CRM 10.2 External data definitions

The Lattice compiler applies a simple rule to external data declarations. If the keyword **extern** is present, the compiler assumes that the actual storage will be allocated elsewhere, and the declaration is a reference to that storage area that will be resolved by the linker. Otherwise, the declaration is interpreted as an actual definition which allocates storage, unless the **-x** option has been used, as described previously under CRM 8.1.

CRM 12.3 Conditional compilation

The constant expression following **#if** may not contain the **sizeof** operator and must appear on a single input line.

CRM 12.4 Line control

Although the filename for **#line** must be an identifier, it need not conform to the characteristics

of C identifiers. The compiler takes whatever string of characters is supplied, and the only lexical requirement for the filename is that it cannot contain any white space.

CRM 14.1 Structures and unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions.

The escape from typing rules described in K&R is explicitly not allowed by the Lattice compiler. In a reference to a structure or union member, the name on the right must be a member of the aggregate named or pointed to by the expression of the left. Our implementation, however, does not enforce any restrictions on references to union members, such as requiring a value to be assigned to a particular member before allowing it to be examined via that member.

4.2 Compiler Implementation Decisions

This section describes how the Lattice compiler deals with some aspects of the C language that have fallen into the domain of "implementation decisions". Some of these issues were left up to the compiler designer simply through oversight in the language definition, and the ANSI committee is gradually closing these loopholes. Others cannot be resolved in the language definition because they depend upon the specific hardware or software limitations faced by the compiler designer.

4.2.1 Pre-Processor Features

The Lattice C compiler supports the full set of pre-processor commands described in K&R. Pre-processor

commands are handled concurrently with lexical and syntactic analysis of the source file, because there is no requirement to have a separate pre-processor pass, and compiler performance is improved by this approach. Nonetheless, analysis of the pre-processor commands is largely independent of the compiler's C language analysis. For example, **#define** text substitutions are not performed for pre-processor commands, but nesting of macro definitions is possible because substituted text is re-scanned for new **#define** symbols.

Since the compiler uses a text buffer of fixed size, a particularly complex macro may occasionally cause a line buffer overflow condition. Usually, however, this error occurs when there is more than one macro reference in the same source line, and it can be circumvented by placing the macros on different lines.

Circular definitions such as:

```
#define A B
#define B A
```

will be detected by the compiler if either A or B is ever used, as will more subtle loops.

Like many other implementations of C, the Lattice compiler pushes macro definitions onto a stack, so that if the line:

```
#define XYZ 12
```

is followed later by:

```
#define XYZ 43
```

the new definition takes effect, but the old one is not forgotten. In other words, after encountering:

```
#undef XYZ
```

the former definition (12) is restored. To completely undefine XYZ, an additional **#undef** is required. The rule is that each **#define** must be matched by a corresponding **#undef** before the symbol is truly forgotten. Also, the compiler issues a warning message whenever a macro is re-defined. This was done to help detect those nasty errors that occur when header files are in conflict.

Two clarifications should be noted with regard to the **#if** command. First, an undefined symbol in a **#if** expression is treated as having a value of zero. Second, a symbol defined with null substitution text is interpreted as having a value of one. These conventions are consistent with **#ifdef** usage, and permit the use of expressions like:

```
#if SYM1 | SYM2 | SYM3
```

which causes subsequent code to be processed if any of the symbols are defined.

4.2.2 Arithmetic Objects

Five types of arithmetic objects are supported by the Lattice compiler. These, as well as pointers, are the basic data entities that can be manipulated by a C program. The types are:

```
short (also, short int)
char
long (also, long int)
float
double (also, long float)
```

In addition, the keyword **unsigned** may be applied as a modifier to any of the integral data types. This

modification does not affect the size of the object, and is significant only when such an object is used in an expression involving conversion, multiplication, division, comparison, or bitwise right shifts.

The natural size of an integer in the AmigaDOS environment is 32 bits. That is, the types **int** and **long** are equivalent. However, if you want your program to be portable to other environments, use **short** when you really want a 16-bit integer, and use **int** when you want a general integer for such things as loop counters.

In expressions involving several data types, the compiler generally carries out the computation in the "widest" data type that is involved. The conversion rules are:

EXPRESSION CONTAINS	**COMPUTATIONAL WIDTH IS**
char	char
short	short
long	long
float	double
double	double

So, for example, if an expression contains items having types **char** and **long**, the **char** types are converted to **long** during the computation.

It is interesting to note that all floating point computations are done in double precision, which is the traditional C standard. This implies that your program will generally execute slower if you use the **float** data type. Many programmers mistakenly believe that **float** data is handled faster than **double**. The ANSI C proposal now allows for computation to be done in single precision floating point if no **double** data items are involved, and we will probably incorporate that feature in

a later release.

4.2.3 Derived Objects

The Lattice C compiler supports the standard extensions leading to various kinds of derived objects, including pointers, functions, arrays, and structures and unions. Declarations of these types may be arbitrarily complex, although not all declarations result in a legal object. For example, arrays of functions or functions returning arrays are illegal. The compiler checks for such illegal declarations and also verifies that structures or unions do not contain instances of themselves.

Objects which are declared as arrays cannot have an array length of zero, unless they are formal parameters or are declared as **extern.**

Note that the size of aggregates (arrays and structures) may be affected by alignment requirements. This topic is discussed later in Section 5.

4.2.4 Storage Classes

Declared objects are assigned by the compiler to storage offsets which are relative to one of several different storage bases. The assigned storage base depends on the explicit storage class specified in the declaration, or on the context of the declaration, as follows:

External — An object is classified as external if the **extern** keyword is present in its declaration, and it is not later declared outside the body of any function without the **extern** keyword.

Storage is not allocated for external

items because they are assumed to exist in some other file, and must be included during the linking process that builds a set of object modules into a load module.

Static An object is classified as static if the **static** keyword is present in its declaration or if it is declared outside the body of any function without an explicit storage class specifier.

Storage is allocated for static items in the data section of the object module. All such locations are initialized to zero unless an initializer expression is included in the declaration.

Static items declared outside the body of any function without the **static** keyword are visible in other files, that is, they are externally defined. Note that string constants are allocated as unnamed static **char** arrays.

Automatic An object is classified as automatic if the **auto** keyword is present in its declaration or if it is declared inside the body of any function without an explicit storage class specifier. It is illegal to declare an **auto** object outside the body of a function.

Storage is allocated for **auto** items on the stack during the execution of the function in which they are defined.

Formal An object is classified as formal if it is a formal parameter to one of the functions in the source file. Storage is

allocated for formal items on the stack by the program that calls the function.

Note that the first phase of the compiler makes no assumption about the validity of the **register** storage class declarator. Items which are declared **register** are so flagged, but storage is allocated for them anyway against either the automatic or the formal storage base.

A register variable declaration may be accepted for any pointer or other data object with a size of no more than 4 bytes. Up to three pointers may be assigned to address registers starting with A4 down through A2; up to six simple data elements may be assigned to data registers starting with D7 down through D2. The registers are assigned in the same order in which they appear in the function declaration, with formal parameters being assigned first. Any additional register variable declarations are accepted by the compiler without being assigned to registers.

The use of register variables affects the entry sequence at the start of the function in which they are declared, by requiring an additional instruction to save the previous registers' values before they are used in the function. See Section 5.3.3 for more information.

Note also that if the **-x** compile-time option is used, the implicit storage class for items declared outside the body of any function changes from **static** to **extern.** This allows a single header file to be used for all external data definitions. When the main function is compiled, the **-x** option is not used, and so the various objects are defined and made externally visible. When the other functions are compiled the **-x** option causes the same declarations to be interpreted as references to objects defined elsewhere.

4.2.5 Scope of Identifiers

The Lattice compiler conforms almost exactly to the scope rules discussed in Appendix A of the Kernighan and Ritchie text (pp. 205-206). The only exception arises in connection with structure and union member names, where, in accordance with later versions of the language, the compiler keeps separate lists of member names for each structure or union. This means that additional classes of non-conflicting identifiers occur for the various structures and unions. Two other points are worth clarifying.

First, the compiler does not generate specific allocate and de-allocate instructions for **auto** items declared in statement blocks within a function. Instead, the function entry sequence allocates enough stack storage to handle the largest collection of automatic data items so declared. With this scheme, a function may allocate more stack space than is actually used, but the need for run-time dynamic allocation within the function is avoided.

Second, when an identifier with a previous declaration is redefined locally as an **extern**, the previous definition is superseded, but the compiler also verifies compatibility with all preceding **extern** definitions of the same name. This is done in accordance with the standards, which require that all references to the same external name must be to the same object. The point is that in this particular case, where a local block redefines an identifier as extern, the local declaration does not actually disappear upon termination of the block because the compiler now has an additional external item for which it must verify later declarations.

4.2.6 Initializers

Objects which are of the static storage class are guaranteed to contain binary zeros when the program begins execution, unless an initializer expression is used to define a different initial value. The Lattice compiler supports the full range of initializer expressions described in Kernighan and Ritchie, but restricts the initialization of pointers somewhat.

A pointer initialization expression must evaluate to the integer constant zero or to a pointer expression of exactly the same type as the pointer being initialized. This pointer expression can include the address of a previously declared static or external object, plus or minus an integer constant. However, it cannot contain a cast operator applied to a variable, because such conversions cannot be done at compile time.

This restriction makes it impossible to initialize a pointer to an array unless the & operator is allowed to be used on an array name, because the array name without the preceding & is automatically converted to a pointer to the first element of the array. Accordingly, the Lattice compiler accepts the & operator on an array name so that declarations such as:

```
int a[5], (*pa)[5] = &a;
```

can be made. Note that if a pointer to a structure (or union) is being initialized, the structure name used to generate an address must also be preceded by the & operator.

An arithmetic object may be initialized with an expression that evaluates to an arithmetic constant which, if not of the appropriate type, is converted to that of the target object.

More complex objects (arrays and structures) may be initialized by bracketed, comma-separated lists of initializer expressions, with each expression corresponding to an arithmetic or pointer element of the aggregate. A closing brace can be used to terminate the list early. See Appendix A of Kernighan and Ritchie for examples.

Unions may also be initialized like structures. The initialization list must correspond to the first member of the union.

A character array may be initialized with a string constant which need not be enclosed in braces. This is the only exception to the rule requiring braces around the list of initializers for an aggregate.

Initializer expressions for auto objects can only be applied to simple arithmetic or pointer types (not to aggregates), and are entirely equivalent to assignment statements.

4.2.7 Expression Evaluation

All of the standard operators are supported by the Lattice compiler, using the standard order of precedence as described on K&R page 49. Expressions are evaluated using an operator precedence parsing technique which reduces complex expressions to a sequence of unary and binary operations involving at most two operands.

Operations involving only constant operands (including floating point constants) are evaluated by the compiler immediately, but no special effort is made to re-order operands in order to group constants. Thus, expressions such as:

```
c - 'A' + 'a'
```

must be parenthesized so that the compiler can evaluate the constant part:

```
c + ('a' - 'A')
```

If at least one operand in a binary operation is not constant, the intermediate expression result is assigned to a temporary storage location. The temporary then replaces the binary operation in the expression and becomes an operand of another binary or unary operation. This process continues until the entire expression has been evaluated.

The use of temporaries is optimized by the compiler so as to minimize re-generation of identical temporaries during a straight-line code sequence. Thus, common sub-expressions are recognized and evaluated only once. For example, in the statement:

```
a[i+1] = b[i+1];
```

the expression **i+1** will be evaluated once and used for both subscripting operations. This same optimization strategy eliminates expressions producing useless results with no other side effects, such as:

```
i+j;
```

Three conditions cause temporaries to "die" during a straight-line code sequence:

1. A function call may have side effects which cause previously-computed temporaries to no longer be accurate, and so all temporaries are discarded after the call.

2. If either operand associated with a temporary is the result of a later operation, the temporary is discarded after that operation.

3. When the result of an operation is stored through a pointer, the compiler discards all temporaries constructed from operands having the same type as the pointer. This is necessary because the compiler cannot determine if the pointer refers to a component of a current temporary value, and so it take the safe approach.

   Note that this strategy may fail if the programmer uses "type punning". For example, if variable A is declared as an integer and you construct a character pointer that, in fact, points to A's storage location, the compiler will not discard temporaries containing A when you change A via that pointer.

Except for common sub-expression detection, which may replace an operation with a temporary value, expressions are evaluated in left-to-right order unless that is prevented by operator precedence or parentheses. However, it is best not to make any assumptions about the order of evaluation, since the language definition allows C compilers to re-order expressions in order to generate better code. We may introduce this type of optimization in a future release.

Note that the language definition does guarantee that logical OR (||) and logical AND (&&) operators will be evaluated in left-to-right order.

4.2.8 Control Flow

C offers a rich set of statement flow constructs, and the Lattice compiler supports the full complement of them. Since control flow operations tend to dominate

many programs, the compiler takes special steps to optimize them, as follows:

1. Switch statements are analyzed to verify that they contain

    1. At least one case or default entry;

    2. No duplicate case values; and

    3. Not more than one default entry.

    Then the compiler chooses the best of several machine code sequences to test for the various cases.

2. The size and number of branch instructions is kept to a minimum by extensive analysis of the flow within each function.

3. Unreachable code is discarded, with the appropriate warning messages.

4.3 Compiler Limitations

To conclude this description of the Lattice's C compiler implementation, we list the more important internal limits of the compiler.

1. The maximum value of the constant expression defining the size of a single subscript of an array is two less than the largest unsigned target machine int (65,533 for a 16-bit int, 4,294,967,293 for a 32-bit int).

2. The maximum length of an input source line is 512 bytes.

3. The maximum size of a string constant is 256 bytes.

4. Macros with arguments are limited to a maximum number of 16 arguments.

5. The maximum length of the substitution text for a **#define** macro is 512 bytes.

6. The maximum level of **#include** file nesting is 1Ø.

These limitations have proven to be adequate for most AmigaDOS C programs, and they also lead to very good compiler performance.

4.4 New Language Features

This section provides simple examples of the use of several of the new language features not described in the Kernighan and Ritchie text.

4.4.1 Void

The reserved word **void** is used to describe a function that has no return value:

```
void noval();              /* function with no return value */
void (*vp)();              /* pointer to void function */
```

While **void** operands may never be used in an expression, it is occasionally useful to cast something to void:

```
int xfunc();               /* function returning int */

(void)(xfunc(a, b));       /* discard return value */
```

The use of the cast shows that the programmer is intentionally discarding the return value from xfunc.

4.4.2 Enumerations

An enumerated data object is integral but may only assume values from a specified list of identifiers, which can be viewed as integer constants. The actual values assigned to the identifiers normally begin at zero and are incremented by one for each successive identifier in the list. An explicit value, however, can be forced for any identifier by using an = assignment. Subsequent identifiers are assigned that new value plus one, and so forth. Here is an example:

```
/* defining an enumerated data type */

enum color {red, blue, green = 4, puce, lavender};

/* defining enumerated data objects and pointers */

enum color x, *px;

/* using enumerated data objects */

x = red;
*px = x;
if (x == lavender) px = &x;
```

In this example, the symbols associated with the enumerated data type color are assigned the following values:

```
0 => red
1 => blue
4 => green
5 => puce
6 => lavender
```

4.4.3 Aggregate Assignment

Structures or unions of identical type may be copied by assignment:

```
struct XYZ
{
int x;
double y;
long z;
} x, y, *px;

x = y;      /* copies contents of y to x */
*px = x;    /* copies x to the pointed to struct */
```

For purposes of assignment or passing to functions (see below), structures of identical type may also appear in a conditional expression:

```
y = (i > 30) ? x : *px;
```

4.4.4 Passing Aggregates by Value

A structure or union which appears in an argument list without a preceding ampersand (&) is passed by value to the function:

```
struct XYZ q;

functn(q);
```

The called function must make an appropriate declaration:

```
void functn(a);
struct XYZ a;
```

Because many existing programs pass the address of a structure without using an ampersand (&) operator, the compiler generates a warning whenever an aggregate is passed by value to a function. Note that the Lattice implementation of aggregate passing by value actually

supplies the called function with a pointer which it uses to make a local copy of the caller's structure immediately upon entry.

4.4.5 Functions Returning Aggregates

A function may return an entire structure or union as a return value:

```
struct XYZ fxyz(a, b, c)
int a;
double b;
long c;
{
struct XYZ r;

r.x = a;
r.y = b;
r.z = c;
return (r);
}
```

The return value must be copied by assignment to another aggregate of the same type:

```
extern struct XYZ x, fxyz();

x = fxyz(2, 20., 2000L);
```

Note that the Lattice implementation actually returns a pointer to a static copy of the returned aggregate. Because this copy persists only long enough to assign the return value, such functions are still recursively reentrant (but not, in general, multi-tasking reentrant).

4.4.6 Function Declarations with Argument Types

External functions may be declared with an enclosed list of type names corresponding to expected arguments. When the function is called, the compiler checks arguments against the types specified in the prototype. Warning messages are issued if the actual argument expressions supplied to the function do not agree with the expected type or number of arguments. If additional arguments may be present for which no type checking is desired, the list of type names may be ended with a comma. Some examples are the following:

```
extern char *sbrk(int);
extern FILE *fopen(char *, char *);
extern double sin(double);
extern void fprintf(FILE *, char *, );
```

Three different warnings may be generated when calls to a function declared with argument types is made. If the number of arguments does not agree with those present in the prototype, warning 87 is issued. If an expression defining an argument does not agree in type with its corresponding declared type name, warning 88 is issued but the expression is not converted to the indicated type. If the expression is a constant, however, warning 89 is issued and the value is converted accordingly.

5. PROGRAMMING ENVIRONMENT

Programming with the Lattice AmigaDOS C Compiler is not much different than programming in C on any UNIX system. This is an important statement, because it implies that you can use just about any of the popular C programming books with the Lattice compiler. Appendix 1 lists some of the publications that you may want to consider.

Now, if you confine your programming to the features that most of these books describe, you really don't need to know much at all about the Amiga environment. However, if you will be doing special things such as linking with assembly-language functions, then you will need to know more about the low-level aspects of the compiler and run-time libraries. The following topics are most important:

Program Sections

Data Representations

Assembly-Language Interfaces

The next subsections discuss each of these topics in turn.

5.1 Program Sections

The compiler isolates you from the details of the 68000 instruction set, but you still need to know a little bit about how your C program gets translated into executable code. The object modules produced by the compiler consist of several sections (what the AmigaDOS manuals call "hunks"), as follows:

CODE Contains the machine instructions that carry out your program operations. The Lattice compiler produces no self-modifying code sequences, so this section can be placed into read-only memory (ROM). The section is named "text" if the **-s** option is used when the module is compiled; otherwise it is unnamed.

DATA Contains all initialized data items, including literals and constants. This section

cannot be placed into ROM because it may contain writable items. The section is named "data" if the **-s** option is used when the module is compiled; otherwise it is unnamed.

BSS Contains all uninitialized data items, which are writable and cannot be placed into ROM. The section is named "udata" if the **-s** option is used when the module is compiled; otherwise it is unnamed. Note that items in this section do not occupy any space in either the object module or the load module; the memory is allocated by the operating system and cleared to zeroes when the program is loaded.

There are two other program sections which are not part of the object file but which make up an important part of the final executing program. The first of these is the stack, and its size is established by the AmigaDOS **stack** command. The default size is 4000 bytes if no **stack** command is used to change the value. The stack is used for context saving during function calls, for parameter passing, and for all automatic storage. Although many C programs use only a few automatic variables, others may require substantially more than 4000 bytes. We recommend that the stack size be increased to at least 10K for running the compiler. You may find it necessary to use even larger values for certain programs, depending on how much automatic storage is used in functions and how much nesting of functions calls occurs.

The second additional section is the heap, also called the memory pool. This is a variable-sized area containing "free memory" that you can allocate and de-allocate via library functions. The amount of memory available for the heap depends on how much system memory is available and also on how much has already been used by the

operating system and other running tasks. The library functions make calls to the operating system to obtain this memory. The requested size is passed directly to the operating system if **sbrk** is used, but if **getmem** is used, the size of the heap is increased by rounding up the requested size according to the value in the external integer **_mstep**. This approach avoids the potentially high cost of the operating system overhead in processing many small allocation requests of AmigaDOS. If your program primarily makes calls to **getmem** for large blocks, a large value for **_mstep** may leave large "holes" in the memory pool. In that case, you can define **_mstep** in your main program with a smaller value.

To summarize, then, the Lattice AmigaDOS environment consists of five distinct memory sections: code, initialized data, uninitialized data, stack, and heap. These normally are scattered throughout the system memory, depending on what memory is available when the program is loaded. If heap space is needed, the memory allocators will request memory from AmigaDOS.

5.2 Data Representations

The compiler maintains data in forms that can be efficiently manipulated by the 68000. Specifically, multibyte data is stored with the high order byte in the lowest address, floating point numbers are kept in the IEEE standard format, and pointers contain 32-bit addresses.

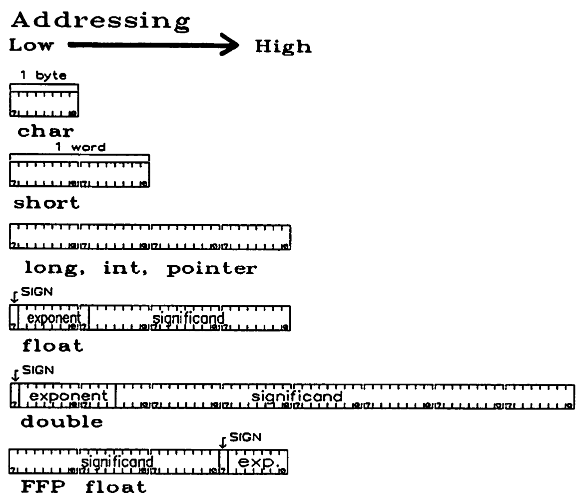
5.2.1 Sizes and Formats

The following table gives the sizes of the C data types.

TYPE	BITS	RANGE
signed char	8	Min: -128 Max: +127
unsigned char	8	Min: 0 Max: 255
signed short int	16	Min: -32,768 Max: +32,767
unsigned short int	16	Min: 0 Max: 65535
signed int	32	Min: -2,147,483,648 Max: +2,147,483,647
unsigned int	32	Min: 0 Max: 4,294,987,295
signed long int	32	Min: -2,147,483,648 Max: +2,147,483,647
unsigned long int	32	Min: 0 Max: 4,294,987,295
float (IEEE format)	32	Small: +/-10E-37 Large: +/-10E+38
double (IEEE format)	64	Small: +/-10E-307 Large: +/-10E+308
pointer	32	Low: 0 High: 0xFFFFFFFF

The more complex C data types, such as arrays and structures, are built up from these basic types. On the 68000, all objects larger than a byte must be accessed from an even address, so structures with single character variables or odd length character arrays may include "holes", i.e., unnamed bytes which are inaccessible but necessary for alignment. These alignment gaps may also be present between elements of an array of structures, if the size of the structure is an odd number of bytes, and between data elements in the same storage class.

The following diagram shows the data type memory formats.



Data Type Memory Formats

5.2.2 Data Portability Issues

On some processors, such as the Intel iAPX88, performance is unaffected by address alignment. On others, such as the Intel iAPX86, while alignment is optional, performance may be improved if special alignment rules are applied to certain of the basic data types. The Motorola MC68000, however, requires the alignment; an attempt to access a 16-bit or 32-bit quantity from an odd address causes a hardware addressing exception.

Note that you should be very careful when exchanging data files between dissimilar systems such as the iAPX86 and the MC68000. For example, consider this short program:

```
#include <stdio.h>
main()
{
FILE *fp;
struct
  {
  char x;
  short y;
  } record;

fp = fopen("testfile","wb");
record.x = 3;
record.y = 4;
fwrite(&record,sizeof(record),1,fp);
fclose(fp);
}
```

If you compile and execute this program under AmigaDOS, the file "testfile" will contain four bytes in the following order: 03 00 00 04. Then if you compile and execute it on a typical iAPX86 system, the file will contain three bytes: 03 04 00. Why? Because first of all, it is standard practice on the MC68000 to align

short integers on even byte addresses; and secondly, the MC68000 normally outputs integers starting from the low-order byte, while the iAPX86 starts with the high-order byte. This all goes to prove that portable programs do not necessarily produce portable data files.

If you are concerned about data file portability, you will save yourself a lot of grief if you think about the problem when designing file formats. The easiest way to handle it is to only write ASCII text files, since text file processing is usually fully portable. If you cannot do this, then you'll need to write a special translation program to re-align arrays and structures and to re-arrange the bytes in integers. These programs are known in UNIX circles as "data swabbers", and they are nasty to write and maintain.

One last comment on data portability: pointers are almost completely non-portable, and so you should almost always avoid writing them to files. In fact, it is generally true that pointers saved to a file are meaningless even when read back into the same computer system by a different program. Why? Because the data being pointed to will not necessarily be in the same place under different execution conditions.

5.3 Assembly-Language Interfaces

As mentioned earlier, it is common practice to develop a program entirely in C and then performance-tune it by re-writing certain functions in assembly language. Also, there are some 68000 operations, such as direct access of special CPU and I/O registers, that are not easily handled in C.

5.3.1 Entry Conditions

Upon entry to a function, the stack will contain all function arguments immediately above the return address, which is 4 bytes long. The arguments appear in left-to-right order; that is, the leftmost argument is immediately above the return address in the stack. All arguments are passed by value. That means, for example, that

```
int i;
double d;

func(i,d);
```

will first push the value of **d** on the stack (8 bytes) and will then push the value of **i** (4 bytes) and will then call **func**. Note that all arguments of type **char** or **short** are promoted to **int** before they are supplied to the called function, so that all arguments in Lattice C are at least 4 bytes in size.

5.3.2 Register Saving

The compiler currently assumes that functions preserve all of the following registers: D2-D7 and A2-A7. Registers A5 and A7 (the stack pointer) must be preserved since they specify the stack frame being used by the calling program. Register A6 must be preserved because it is expected to contain the data section base value. The other registers in this list must be preserved because they may be in use by the calling program as register variables. Registers DØ and D1 are used for function return values and do not need to be preserved. Registers AØ and A1 are not used for register variables, and the compiler does not assume that their contents are preserved.

5.3.3 Return Values

Function return values are passed back in one or more registers, depending on the data type declared for the function. The conventions are:

char	Value is returned in DØ.B (low byte of DØ).
short int	Value is returned in DØ.W (low word of DØ).
long int	Value is returned in DØ.
float	Value is returned in DØ.
double	Value is returned in DØ and D1, with the high-order part in DØ.
pointer	Value is returned in DØ.

Note that functions returning aggregates (structures or unions) actually return a pointer to a static copy of the aggregate. Because this copy persists only long enough to assign the return value, such functions are recursively reentrant but not multi-tasking reentrant.

5.3.4 Writing Assembly-Language Modules for C

You can write assembly language modules for inclusion in C programs if you are sufficiently familiar with the 68ØØØ instruction set and if you observe the following:

1. The statements defining the functions should be placed in the text section by preceding them with

```
CSECT     text
```

This is not an absolute requirement, since the functions will be accessible regardless of the section in which they are defined, but it assures them of placement with the C functions during linking (if the -s option is used). Each function entry must be declared in an XDEF statement:

```
        XDEF    AFUNC
          .
          .
          .
AFUNC   EQU     *
```

2. If the module is to define data locations to be accessed (using extern declarations) in C modules, those definitions should be placed in the data section by preceding them with

```
        CSECT   data
```

   Each data element must be declared in an XDEF statement in order to be accessible in the C modules:

```
        XDEF    DX,DY,DZ
DX      DC.L    4000H
DY      DC.W    8000H
DZ      DC.L    DX
```

3. Any of the registers D2-D7 or A2-A6 must be preserved by the module, and the return value loaded into the appropriate data registers.

To call a C function from an assembly language module, you must include an XREF declaration for the function. Before calling the function (via JSR), you must supply any expected arguments in the proper order. After con-

trol returns from the called function, the stack pointer must be adjusted to account for pushed arguments.

```
XREF      cfunc
MOVE.L    D0,-(A7)        push argument
MOVE.L    D1,-(A7)
JSR       cfunc           call function
ADDQ      #8,A7           restore stack ptr
```

Data elements defined in a C module may be accessed via XREF statements, as well:

```
XREF      XD2,XD3
.
.
.
MOVE.L    XD2,D0
```

5.4 Addressing Modes

The compiler provides options that allow variations in the way data and functions are referenced. These options can be used to reduce load module sizes or improve efficiency. Although there can be disadvantages, significant improvements can often be achieved through judicious use of the appropriate options.

One of the more notable effects of these options is the elimination of a large amount of relocation information from the load module. This can be very significant for programs with a large number of small functions.

5.4.1 Base Relative Addressing

The **-b** option directs the compiler to reference static data as a 16-bit offset from the base register A6 in-

stead of the normal 32-bit direct reference. This form of addressing has the advantage of requiring only 4 bytes per instruction instead of the 6 bytes necessary for a direct reference. These instructions also generally execute faster.

However, if the **-b** option, or base-relative addressing mode, of the compiler is used, it is necessary to combine some or all of the data sections. Since this addressing mode references data locations as offsets from the contents of A6, this means that the data must reside in a contiguous block instead of the normal scattered distribution. Also, because the offset field of the instruction is only 16-bits, this limits the amount of data that can be referenced this way to 64K.

The compiler automatically names the data and uninitialized data sections of modules compiled with the **-b** option as **__MERGED**, overriding any names for these sections that may have been specified with the **-s** flag. The linker will automatically merge sections with this name into the data section referenced via register A6.

It is possible to mix modules compiled with and without the **-b** flag. As long as modules compiled with the **-b** flag do not reference any data in modules compiled without the **-b** flag, no special linking procedures are necessary. Otherwise, it is necessary to use the **smalldata** option of the linker to cause all data to be combined.

There is one important exception to this. The phase 2 option **-c** can conflict with the **-b** option. If the **-c** option has been used to specify that the data sections are to be loaded into chip memory, the linker will not merge those sections into the special A6 addressable data section. Therefore, data compiled with the phase 2 option **-c** should never be referenced by modules compiled with the **-b** option. If the **-b** option is used with the

phase 2 option **-c**, the compiler will display a warning and revert to the normal 32-bit addressing mode during code generation.

If a program compiled with **-b** will be used as an interrupt handler or with the **AddTask** function, you must compile with the **-y** option so that the program will automatically load A6 with the linker supplied value for the base of the data section. This is necessary because such a task will not be performing any of the initialization normally done by the startup routine **c.o**, and therefore would not have a correct value in A6.

5.4.2 PC-Relative Branches

Normally the compiler generates 32-bit function references when a program calls an externally defined function. The **-r** option will cause all function calls to be PC-relative, or relative to the current location counter. These instructions are 4 bytes long as opposed to the 6 bytes required for a direct reference. They also execute faster. However, to be effective the target location, i.e. the function being called, must be within +/-32K of the branch instruction. If the target location is beyond this range, either because it is in another segment or the current segment is too large, **blink** must adjust the instruction to branch through a linker generated jump table. This can increase the amount of time required to perform the branch.

6. REFERENCE SECTIONS

After the Appendices, the remainder of this manual is divided into lettered reference sections, as follows:

Section C: Commands
Describes the AmigaDOS commands that are

provided with the compiler package.

Section D: Data Names
Describes the public data names that are defined in the Lattice libraries.

Section E: Environment Variables
Describes the AmigaDOS environment variables used by the compiler and related programs.

Section F: Function Names
Describes the public function names that are defined in the Lattice libraries.

Section X: Combined Index
An alphabetical listing of all commands and external names. Each individual section is also preceded by a name index for that section.

Some general information on each of these sections is presented below.

6.1 Commands

Section C describes the AmigaDOS commands that are used to invoke the compiler and the various utility programs.

When you install the compiler, the driver program **lc** is usually copied into the command directory C:. Most operations, including compiling, library updating and linking can be performed via the **lc** command. Most people would then add the appropriate assignment statements to their **startup-sequence** file in the S: directory so that **lc** can easily locate all the programs and files it requires. See Section E for descriptions of the logical name assignments used by **lc**.

Commands are invoked in the following way:

```
name <input >output options arguments
```

where all fields except **name** are optional. The fields are defined as follows:

name
: This is the command name. AmigaDOS makes no distinction between upper and lower case in this field.

<input
: This field, if present, specifies the file that is to be assigned as this command's standard input. Within a Lattice C program, the standard input file is specified by the Level 2 I/O file pointer named **stdin.** The **input** field can be any valid AmigaDOS file or device reference, and no distinction is made between upper and lower case.

>output
: This field, if present, specifies the file that is to be assigned as this command's standard output. Within a Lattice C program, the standard output file is specified by the Level 2 I/O file pointer named **stdout.**

options
: This optional field contains one or more "option specifiers" with blank separators. In its simplest form, an option specifier consists of a dash (-) followed by a single letter. For example, the **-v** option on the **lc** command tells the compiler not to generate stack-checking code. Some options contain variable information, which should immediately follow the option letter. For example, **-oDF1:** on the **lc** command tells the compiler to place the object file on disk drive **DF1:**.

Note that some commands are sensitive to the case of the option letter. Also note that our commands use options in the UNIX way, with leading dashes.

arguments This field, if present, consists of one or more strings that will be passed to your main program via the **argv** array, as described in Section F. The arguments are separated by white space. Any argument containing white space must be enclosed in double quotes, and you can use \" to include a double quote within such an argument.

6.2 Data and Function Names

Sections D and F describe the external names (i.e. data and functions) that you will probably need to access from your C programs.

In general, you should not define data and function names that are the same as any of these, because your names will override those in the libraries. And since the library calls upon itself, your override may interfere with proper library operation. For example, if you replace the **strcmp** function with one that behaves differently, the **tqsort** function will probably not work, because it calls **strcmp**. Note that if you do duplicate one of the Lattice external names, the linker will not usually report this as an error.

6.2.1 Portability Classifications

Most C programmers are somewhat concerned about portability, in the sense that they do not want to write

software that locks them into a particular computer. Also, C programmers like to design general-purpose modules (i.e. "software tools") that they can re-use from one application to another. At the same time, C is popular because it allows you to write programs approaching the performance of assembly language but with less effort and fewer problems.

The Lattice AmigaDOS library attempts to strike a reasonable balance between portability and performance. It includes the highly portable functions defined by the ANSI C Standards Committee, and if you use only those, your program will probably run correctly on a broad variety of machines.

The library also includes a large number of functions that utilize AmigaDOS more efficiently than the ANSI functions or that provide access to features that are peculiar to AmigaDOS.

It's up to you to decide which of these facilities make sense for your application. To assist you in this decision, each data or function name has been given a portability classification, as follows:

ANSI The name is defined in the proposed ANSI Standard for the C language, and our implementation conforms to that definition. Lattice is an active member of the ANSI committee, and we resolve any compatibility issues in that forum.

UNIX The name is defined in AT&T's UNIX System V, and our implementation conforms to that definition. We use UNIX on a VAX computer to verify compatibility.

XENIX The name is defined in Microsoft's XENIX, and our implementation conforms to that

definition. We use XENIX on an IBM-AT computer to verify compatibility.

LATTICE The name is defined in Lattice's portable library. That is, we guarantee that it will be available on any system for which we provide a C compiler package.

AMIGA The name is defined by Lattice, but it is only available under AmigaDOS.

6.2.2 Function Calling Sequences

It is very important that you understand how we are presenting function calling sequences. In the synopsis section of each function description, we show the general form of the function call, such as:

```
fh = open(name,mode,prot);
```

Then we indicate what kind of information the function expects to receive as its arguments and pass back as its return value. For the **open** function, this is expressed by:

```
int fh;          file handle
char *name;      file name
int mode;        access mode
int prot;        protection mode
```

You, as the caller, must supply arguments that conform to the function's expectations. Novice C programmers sometimes misinterpret our presentation and conclude that they must declare their arguments exactly as we show in the synopsis. This has led to programs like:

```
int fh,mode,file,prot;
char *name;
name = "myfile";
mode = O_RDONLY;
prot = 0;
fh = open(name,mode,prot);
file = fh;
```

which is wasteful and unnecessary. A much more straightforward approach would be:

```
int file;
file = open("myfile",O_RDONLY,0);
```

In other words, the arguments can be expressed any way you like, as long as they arrive at the function in the correct form. And, of course, the function neither knows nor cares about the name you give its return value.

However, in order for the return value to be handled properly, you must make sure that the function is declared correctly. For example, this sequence of code will produce bizarre results, assuming that there is no declaration for the **sin** function:

```
double value;

value = sin(1.7);
```

Why? Because in the absence of a declaration, the compiler assumes that **sin** returns an integer, and so code is generated to convert an integer result into a double for assignment to **value**. Since **sin** actually returns a double, the conversion is unnecessary and produces garbage. The correct code is:

```
double value,sin();

value = sin(1.7);
```

This alerts the compiler that **sin** returns a double, which does not then need to be converted for the assignment.

An even better way to ensure that the standard Lattice functions get declared properly is to include the appropriate header files into your program. Whenever a special header file should be used in conjunction with a particular function or variable, we show the appropriate **#include** statement in the synopsis. The header file will contain any definitions (e.g. constants and structures) that are normally required when you interface with the function or variable. Also, it will contain the necessary external declarations, as discussed next.

6.2.3 Argument Type Checking

The compiler can check that functions are called with appropriate types of arguments. This feature is automatically activated when you declare an external function with an argument type list, such as:

```
extern int read(int, char *, unsigned);
```

which indicates that the **read** function should be called with three arguments: an integer, a character pointer and an unsigned integer.

The various header files contain such declarations for the standard Lattice library functions. However, these declarations can be bypassed if you define the preprocessor variable **NARGS** before including any header files. This can be done on the **lc** or **lc1** command line as

```
lc -dNARGS myprog
```

or you can include the line

```
#define NARGS
```

at the beginning of your source file. Note that **NARGS** must be capitalized.

6.2.4 File Names

For all functions that accept file name arguments, you can use the "complete path" form unless explicitly noted otherwise. The complete path form is:

```
drive:path/node.ext
```

where **drive** is a drive or volume name, **path** is a directory path, **node** is a node name, and **ext** is an extension. The directory path can be a list of **node.ext** constructs separated by slashes. Lattice functions that generate or parse file names use the character defined in **_SLASH** as the separator. This character defaults to a slash.

Here are several examples of valid file names:

```
myprog
myfile.c
df1:myprog
/pdq/myprog
df0:/pdq/xyz.dir/myfile.c
```

6.3 Environment Variables

Section E describes the AmigaDOS logical name assignments that are used by the Lattice compiler and related programs.

Initially, all logical name assignments are constructed by AmigaDOS when the system is first loaded. For example, **C:** is the logical name assigned to the command directory, and **LIBS:** is the logical name assigned to the directory containing run-time libraries. Additional assignments can be made with the **assign** command. Logical names can be assigned to any disk volume, directory or file. The most popular use for logical names is to specify the directories where certain file types can be found. They can also be used as shorthand references to the files themselves. For instance, assigning **CC:** to the file **DF1:lc** allows you to refer to the file **lc** on the volume in drive **DF1:** as **CC:**.

APPENDIX 1
BIBLIOGRAPHY

This appendix lists some sources of additional information about AmigaDOS and C programming. This list is not exhaustive nor does it constitute any specific endorsement by Lattice, Inc.

C PROGRAMMING REFERENCES:

Bolsky, M.I. [1985]
"The C Programmer's Handbook", ISBN 0-13-110073-4, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Brand, Kim Jon [1985]
"Common C Functions", ISBN 0-88022-069-4, QUE Corporation, Indianapolis, IN 46250.

Gehani, Narain [1985]
"Advanced C: Food For The Educated Palate", ISBN 0-88175-078-6, Computer Science Press, Rockville, MD 20850.

Harbison, Samuel P. and Guy L. Steele Jr [1984]
"A C Reference Manual", ISBN 0-13-110008-4, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Hogan, Thom [1984]
"The C Programmer's Handbook', ISBN 0-89303-365-0, Brady Communications Company, Inc., Bowie, MD 20715.

Hunt, William James [1985]
"The C Toolbox", ISBN 0-201-11111-X, Addison-Wesley Publishing Company, Reading, MA.

Kernighan, Brian W. and Dennis M. Ritchie [1978]
"The C Programming Language", ISBN 0-13-110163-3,Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Plum, Thomas and Jim Brodie [1985]
"Efficient C", ISBN 0-911537-05-8, Plum Hall, Cardiff, NJ 08232.

Plum, Thomas [1983]
"Learning to Program in C", ISBN 0-911537-00-7, Plum Hall, Cardiff, NJ 08232.

Purdum, Jack [1983]
"C Programming Guide", ISBN 0-88022-022-8, QUE Corporation, Indianapolis, IN 46250.

Schustack, Steve [1985]
"Variations In C", ISBN 0-914845-48-9, Microsoft Press, Bellevue, WA 98009.

Schwaderer, W. David [1985]
"C Wizard's Programming Reference", ISBN 0-471-82641-3, Wiley Press, New York, NY 10158.

Sobelman, Gerald E. and David E. Krekelberg [1985]
"Advanced C Techniques & Applications", ISBN 0-88022-162-3, Que Corporation, Indianapolis, IN 46250.

Traister, Robert J. [1985]
"Going from BASIC to C",ISBN 0-13-357799-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Traister, Robert J. [1984]
"Programming In C For The Microcomputer User", ISBN 0-13-729641-X, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

AmigaDOS References:

Commodore-Amiga, Inc. [1985]

"AmigaDOS User's Manual", Commodore Business Machines, Inc., West Chester, PA 19380

Commodore-Amiga, Inc. [1985]

"AmigaDOS Technical Reference Manual", Commodore Business Machines, Inc., West Chester, PA 19380

Commodore-Amiga, Inc. [1985]

"AmigaDOS Developer's Manual", Commodore Business Machines, Inc., West Chester, PA 19380

Commodore-Amiga, Inc. [1985]

"Amiga ROM Kernel Manual, Vols 1 & 2", Commodore Business Machines, Inc., West Chester, PA 19380

Commodore-Amiga, Inc. [1985]

"Amiga Intuition Manual", Commodore Business Machines, Inc., West Chester, PA 19380

Commodore-Amiga, Inc. [1985]

"Amiga Hardware Manual", Commodore Business Machines, Inc., West Chester, PA 19380

Commodore-Amiga, Inc. [1986]

"The AmigaDOS Manual", Bantam Books, Inc., 666 Fifth Avenue, New York, NY 10103

Compute! Publications, Inc. [1986]

"Compute!'s Amiga Programmer's Guide", ISBN 0-87455-028-9, Compute! Publications, Inc., P.O. Box 5406, Greensboro, NC 27403

Compute! Publications, Inc. [1986]

"Inside Amiga Graphics", ISBN 0-87455-040-8, Compute! Publications, Inc., P.O. Box 5406,

Greensboro, NC 27403

Mortimore, Eugene P. [1986]
"Amiga Programmer's Handbook", ISBN 0-895880-343-0,
SYBEX, Inc., 2344 Sixth Street, Berkely, CA 94710

MC68000 REFERENCES

Kane, G., D. Hawkins and L. Leventhal [1981]

"68000 Assembly Language Programming", ISBN 0-931988-62-4, Osborne/McGraw-Hill, 2600 Tenth Street, Berkely, CA 94710

King, Tim and Brian Knight [1983]

"Programming the M68000", ISBN 0-201-14635-5, Addison-Wesley Publishing Company, Reading, MA.

Motorola Inc. [1984]

"The M68000 Programmer's Reference Manual", ISBN 0-13-566795-X, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632

Scanlon, Leo J. [1981]

"The 68000: Pinciples and Programming", ISBN0-672-21853-4, Howard W. Sams & Co., 4300 West 62nd St., Indianapolis, IN 46268

Williams, Steve [1985]

"Programming the 68000", ISBN0-89588-133-0, SYBEX, Inc., 2344 Sixth Street, Berkely, CA 94710

APPENDIX 2
DISTRIBUTION DISKETTE FILE LIST

This appendix lists the files that you should find on the distribution diskettes. See the READ.ME file on diskette #1 for any recent changes, additions or deletions.

This appendix lists the contents of the two disks that comprise the Lattice AmigaDOS C Compiler package. The first disk is a bootable system disk containing the executable programs. The second disk contains the Lattice and Amiga header files and libraries, source files for some of the routines in the Lattice run-time support library, and some sample programs to use as examples of programming in C on the Amiga.

FILE CONTENTS: DISK #1

```
*               Disk #1
*
* The C directory contains all the executable programs and commands
* Descriptions of most of these programs can be found in the
* AmigaDOS User's Manual.
*
c                        command program directory
c/Assign
c/asm                    Lattice 68000 assembler
c/blink                  Lattice amiga linker program
c/Break
c/CD
c/Copy
c/Date
c/Delete
c/Dir
c/DiskCopy
c/Echo
c/Ed
c/Else
c/EndCLI
c/EndIf
c/Execute
c/FailAt
c/Fault
c/FileNote
c/Format
```

```
c/If
c/Info
c/Install
c/Join
c/Lab
c/lc                     Lattice compiler driver program
c/lc1                    Lattice compiler phase 1 program
c/lc2                    Lattice compiler phase 2 program
c/List
c/LoadWb
c/MakeDir
c/NewCLI
c/omd                    Lattice object module disassembler program
c/oml                    Lattice object module librarian program
c/Prompt
c/Protect
c/Quit
c/Relabel
c/Rename
c/Run
c/Search
c/Skip
c/Sort
c/Stack
c/Status
c/Type
c/Wait
c/Why
*
* The DEVS directory contains the non-resident system device drivers
*
devs                     device driver directory
devs/clipboard.device
devs/parallel.device
devs/printers            directory containing various printer drivers
devs/printers/alphacom_pro101
devs/printers/brother_hr-15xl
devs/printers/cbm_mps1000
devs/printers/diablo_630
```

```
devs/printers/diablo_adv_d25
devs/printers/diablo_c-150
devs/printers/epson
devs/printers/epson_jx-80
devs/printers/generic
devs/printers/hp_laserjet
devs/printers/hp_laserjet_plus
devs/printers/okimate_20
devs/printers/qume_ltrpro_20
devs/serial.device
devs/system-configuration
*
Disk.info
*
* The EMPTY directory (or drawer) is used to create new directories
* from Workbench by selecting, then duplicating
*
Empty                   empty directory
Empty.info              icon for empty directory
*
* The L directory contains system functions
*
l                       system library directory
l/Disk-Validator
l/Port-Handler
l/Ram-Handler
*
* The LIBS directory is for transient run-time libraries
*
libs                    system run-time library directory
libs/diskfont.library
libs/icon.library
libs/info.library
libs/mathieeedoubbas.library
libs/mathtrans.library
libs/version.library
*
on Lattice C            program to display contents of "read.me"
on Lattice C.info       icon for "on Lattice C" program
```

```
*
Preferences             program to set screen resolution, color, etc.
Preferences.info        icon for "Preferences"
*
read.me                 text file of the latest compiler changes
*
* The S directory is where batch files (or 'procedures') are normally
* located
*
s                       directory containing batch files
s/startup-sequence      system startup batch file
s/install_hd            compiler hard-disk installation procedure
*
setdate                 program to set the system date/time
*
* The SYSTEM directory contains the disk utilities and CLI
*
System                  directory containing system utilites
System.info             icon for "System" directory
system/.info
system/CLI              program to create a CLI task
system/CLI.noinfo       icon for CLI program
system/DiskCopy         diskcopy utility program
system/DiskCopy.info    icon for diskcopy
system/Initialize       disk intialization program
system/Initialize.info  icon for initialization program
*
* The T directory is used by Amiga programs for storing 'temporary'
* files
*
t                       directory for temporary files (used by "ed")
*
* The TRASHCAN directory is where items to be deleted are placed
*
Trashcan                Workbench disposal directory
Trashcan.info           icon for "Trashcan"
trashcan/.info
```

FILE CONTENTS: DISK #2

```
*
* The following are the Lattice header files that are part of every
* Lattice C compiler package.
*
assert.h        Assert macro
ctype.h         Character type macros
dos.h           DOS specific information
error.h         Error codes for "errno"
fcntl.h         Level 1 I/O information
fctype.h        Character type functions
float.h         Floating point information
ios1.h          Additional Level 1 I/O information
limits.h        Non-float limits
math.h          Additional floating point info
setjmp.h        Definitions for "setjmp"
signal.h        Definitions for "signal"
stdio.h         Level 2 I/O information
stdlib.h        Definitions for standard library functions
string.h        Definitions for string functions
time.h          Definitions for "time" et al
*
* The following subdirectories and header files are Amiga specific, and
* are provided by Commodore-Amiga.
*
clib
clib/macros.h
devices
devices/audio.h
devices/bootblock.h
devices/clipboard.h
devices/console.h
devices/conunit.h
devices/gameport.h
devices/input.h
devices/inputevent.h
devices/keyboard.h
devices/keymap.h
```

devices/narrator.h
devices/parallel.h
devices/printer.h
devices/prtbase.h
devices/serial.h
devices/timer.h
devices/trackdisk.h
exec
exec/alerts.h
exec/devices.h
exec/errors.h
exec/exec.h
exec/execbase.h
exec/execname.h
exec/interrupts.h
exec/io.h
exec/libraries.h
exec/lists.h
exec/memory.h
exec/nodes.h
exec/ports.h
exec/resident.h
exec/tasks.h
exec/types.h
graphics
graphics/clip.h
graphics/collide.h
graphics/copper.h
graphics/display.h
graphics/gels.h
graphics/gfx.h
graphics/gfxbase.h
graphics/gfxmacros.h
graphics/graphint.h
graphics/layers.h
graphics/rastport.h
graphics/regions.h
graphics/sprite.h
graphics/text.h

```
graphics/view.h
hardware
hardware/adkbits.h
hardware/blit.h
hardware/cia.h
hardware/custom.h
hardware/dmabits.h
hardware/intbits.h
intuition
intuition/intuition.h
intuition/intuitionbase.h
libraries
libraries/diskfont.h
libraries/dos.h
libraries/dosextens.h
libraries/mathffp.h
libraries/translator.h
resources
resources/cia.h
resources/disk.h
resources/misc.h
resources/potgo.h
workbench
workbench/icon.h
workbench/startup.h
workbench/workbench.h
*
* The EXAMPLES directory contains several C programs that may be used
* as examples of various Amiga operations.
* This directory will change as more or better examples are found.
*
examples
examples/blines.c       sample line drawing program
examples/drag.c         sample fractal program
examples/drag.h
examples/setdate.c      program to get/set system time and date
examples/speechtoy.c    program to demo speech, graphics, gadgets
*
* The LIB directory contains the Lattice and Amiga run-time support
```

```
* libraries, the startup routine 'c.o', and a version of the low-level
* I/O interface that performs stack checking
*
lib
lib/amiga.lib           Amiga's run-time support library interface
lib/c.o                 Lattice startup routine
lib/iosØ.o              Lattice low-level I/O modules
lib/lc.lib              Lattice run-time support library
lib/lcm.lib             Lattice IEEE math library
lib/lcmffp.lib          Lattice interface to Amiga's FFP math library
*
* The SOURCE directory contains source to various parts of the Lattice
* run-time support library that can be customized by knowledgeable
* programmers
*
source
source/c.a              primary initialization routine
source/cxbrk.c          default break character handler
source/cxferr.c         low-level floating point error exit
source/cxovf.c          stack overflow handler  (part 2)
source/ffptran.a        interface to Amiga's FFP trig functions
source/matherr.c        high-level floating point error exit
source/oserr.c          operating system errors
source/syserr.c         UNIX error messages
source/_assert.c        Assert failure exit
source/_cxovf.a         stack overflow handler (part 1)
source/_main.c          secondary initialization routine
```

APPENDIX 3
COMPILER ERROR MESSAGES

This appendix lists the various error messages produced by the Lattice AmigaDOS C Compiler.

TABLE OF CONTENTS

1. OVERVIEW

The Lattice AmigaDOS C Compiler produces three types of error messages:

Operational Errors

These indicate that the compiler is having trouble operating correctly because it cannot access required files or cannot obtain enough disk or memory space.

Syntax Errors and Warnings

These indicate that the compiler is having difficulty understanding your C program. The message includes the source file name and line number indentifying the point at which the problem was detected. An "error message" indicates that the problem prevents the construction of a usable object module and must, therefore, be corrected. A "warning message" indicates that the compiler detected something unusual but will proceed to make an object module, using appropriate assumptions about what you intended the source code to do.

Internal Errors

These indicate that the compiler encountered some internal condition that should not have occurred. This is the old "I shouldn't be here" type of message, and you should report it to our Technical Support Department. However, before doing so, we would be grateful if you would conduct a few experiments with your source code to see if you can make the problem go away. The internal error explanations given below should provide enough clues.

2. OPERATIONAL ERRORS

Can't open source file

The first phase of the compiler was unable to open the source file. This error usually occurs because you mispelled the file name or did not specify the proper drive and/or directory path.

Can't create object file

The second phase of the compiler was unable to create the object file. This error usually results from a full directory on the output disk.

Can't create quad file

The first phase of the compiler was unable to create the quad file. This error usually results from a full directory on the output disk.

Can't open quad file

The second phase of the compiler was unable to open the quad file. This error usually occurs when you call the LC1 command directly with an invalid quad file name.

Combined output file name too large

The output file name constructed by combining the source or quad file name with the text specified using the **-o** option exceeded the maximum file name size of 64 bytes.

File name missing

A required file name was not specified. For the LC and LC1 commands, you must specify a source file, and LC2 requires a quad file name.

Intermediate file error

The first phase of the compiler encountered an error when writing to the quad file. This error

usually results from an out-of-space condition on the output disk.

Invalid command line option
An invalid command line option was specified, and that option will be ignored. See **lc.exe, lc1.exe** and **lc2.exe** in Section C for valid command line options.

Invalid symbol definition
The name attached to **-d** specifying a symbol to be defined was not a valid C identifier or was followed by text which did not begin with an equal sign.

No functions or data defined
The compiler reached the end of the source file without finding any data or function definitions. One common cause of this error is to forget a comment terminator (*/) during the first function in the source file. This causes the compiler to gobble up your program as if it were a comment.

Not enough memory
This message is generated when either phase of the compiler uses up all the available working memory. To cure this, you can switch to the "big compiler" via the **-B** option on the **lc.exe** command. If you are already using the big compiler, then you will need to install more memory or remove some resident service programs (e.g. keyboard enhancers or desk managers) from your system. An alternative is to break your source file into smaller pieces. This is easy if the file consists of a lot of small functions. If it consists of a few extremely large functions, you should probably ask yourself if you are making good use of C's excellent support for modular programming.

Object file error

The second phase of the compiler encountered an error when writing to the object file, probably because there is no more space on the output disk.

Parameters beyond file name ignored

Additional information was present on the LC1 or LC2 command line beyond the name of the source or quad file to be compiled. A common source of this error is to place compiler options after the file name, which was required on earlier versions of Lattice C.

Unrecognized -c option

One of the characters following the **-c option was not** a recognized compiler control character. See the **lc.exe** and **lc1.exe** commands for a list of the valid compiler control options.

-i option ignored

More than four **-i** option strings were specified. Only the first four are retained and used.

3. SYNTAX ERRORS AND WARNINGS

Syntax errors and warnings are reported via a message with the following format:

```
fff nnn Error xxx: mmm
```

where the message components are:

fff This is the name of the source file that was being processed when the error occurred.

nnn This is the number of the source file line that was being scanned when the error occurred. Source file lines begin at 1, not Ø.

xxx This is the error number, as listed below.

mmm This is the error message, as listed below.

Note that all of these fields are variable length.

All messages indicate "fatal errors" unless the message number listed below is followed by **(W)**. When a fatal error occurs, the compiler will not produce a usable object module. The **lc.exe** command alerts you to this condition by beeping and pausing, unless you use the **-C** option to force continuous compilation.

If the message number below is followed by **(W)**, then it is a warning. When such a message is displayed, the compiler will produce a usable object module by making reasonable assumptions about what you intended the source file to do. Nonetheless, it's a good idea to investigate these warnings, since the compiler's assumptions may disagree with your intentions.

1 This error is generated by a variety of conditions in connection with pre-processor commands, including specifying an unrecognized command, failure to include white space between command elements, or use of an illegal pre-processor symbol.

2 The end of an input file was encountered when the compiler expected more data. This may occur on an #include file or the original source file. In many cases, correction of a previous error will eliminate this one.

3 The file name specified on an #include command was not found.

4 An unrecognized element was encountered in the input file that could not be classified as any of the valid lexical constructs (such as an identifier or one of the valid expression operators). This may occur if control characters or other illegal characters were detected in the source file.

5 A pre-processor #define macro was used with the wrong number of arguments.

6 Expansion of a #define macro caused the compiler's line buffer to overflow. This may occur if more than one lengthy macro appeared on a single input line.

7 The maximum extent of #include file nesting was exceeded; the compiler supports #include nesting to a maximum depth of 4.

8 A cast (type conversion) operator was incorrectly specified in an expression.

9 The named identifier was undefined in the context in which it appeared, that is, it had not been previously declared. This message is only generated once; subsequent encounters with the identifier assume that it is of type int (which may cause other errors).

10 An error was detected in the expression following the [character (presumably a subscript expression). This may occur if the expression in brackets was null (not present).

11 The length of a string constant exceeded the maximum allowed by the compiler (256 bytes). This will occur if the closing " (double quote) was omitted in specifying the string.

12 The expression preceding the . (period) or -> structure reference operator was not recognizable by the compiler as a structure or pointer to a structure.

13 An identifier indicating the desired aggregate member was not found following the . (period) or -> operator.

14 The indicated identifier was not a member of the structure or union to which the . (period) or -> referred.

15 The identifier preceding the (function call operator was not implicitly or explicitly declared as a function.

16 A function argument expression specified following the (function call operator was invalid. This may occur if an argument expression was omitted.

17 During expression evaluation, the end of an expression was encountered but more than one operand was still awaiting evaluation. This may occur if an expression contained an incorrectly specified operation.

18 During expression evaluation, the end of an expression was encountered but an operator was still pending evaluation. This may occur if an operand was omitted for a binary operation.

19 The number of opening and closing parentheses in an expression was not equal. This error message may also occur if a macro was poorly specified or improperly used.

20 An expression which did not evaluate to a constant was encountered in a context which required a constant result. This may occur if one of the operators not valid for constant expressions was present.

21 An identifier declared as a structure or union was encoun tered in an expression where aggregates are not permitted. Only the direct assignment and conditional operators may be used on aggregates, and explicit or implicit testing of aggregates as a whole is not permitted.

22 (W) An identifier declared as a structure or union appeared as a function argument without the preceding & operator. In Version 3 of Lattice C, aggregates may be passed by value, so this is a legal construction. The warning message is generated to alert you that earlier versions of Lattice C passed the address of the aggregrate in this case.

23 The conditional operator was used erroneously. This may occur if the ? operator was present but the : was not found when expected.

24 The context of the expression required an operand to be a pointer. This may occur if the expression following * did not evaluate to a pointer.

25 The context of the expression required an operand to be an lvalue. This may occur if the

expression following & was not an lvalue, or if the left side of an assignment expression was not an lvalue.

26 The context of the expression required an operand to be arithmetic (not a pointer, function, or aggregate).

27 The context of the expression required an operand to be either arithmetic or a pointer. This may occur for the logical OR and logical AND operators.

28 During expression evaluation, the end of an expression was encountered but not enough operands were available for evaluation. This may occur if a binary operation was improperly specified.

29 An operation was specified which was invalid for pointer operands (such as one of the arithmetic operations other than addition).

30 (W) In an assignment statement defining a value for a pointer variable, the expression on the right side of the = operator did not evaluate to a pointer of the exact same type as the pointer variable being assigned, i.e., it did not point to the same type of object. The warning also occurs when a pointer of any type is assigned to an arithmetic object. Note that the same message may be a fatal error if generated for an initializer expression.

31 The context of an expression required an operand to be integral, i.e., one of the integer types (char, int, short, unsigned, or long).

32 The expression specifying the type name for a cast (conversion) operation or a sizeof expression was invalid.

33 An attempt was made to attach an initializer expression to a structure, union, or array that was declared auto. Such initializations are expressly disallowed by the language.

34 The expression used to initialize an object was invalid. This may occur for a variety of reasons, including failure to separate elements in an initializer list with commas or specification of an expression which did not evaluate to a constant. Some experimentation may be required in order to determine the exact cause of the error.

35 During processing of an initializer list or a structure or union member declaration list, the compiler expected a closing right brace, but did not find it. This may occur if too many elements were specified in an initializer expression list or if a structure member was improperly declared.

36 A statement within the body of a switch statement was not preceded by a case or default prefix which would allow control to reach that statement. This may occur if a break or return statement is followed by any other statement without an intervening case or default prefix.

37 The specified statement label was encountered more than once during processing of the current function.

38 In a body of compound statements, the number of opening left braces { and closing right

braces } was not equal. This may occur if the compiler got "out of phase" due to a previous error.

39 One of the C language reserved words appeared in an invalid context (e.g., as a variable name). Note that entry is reserved although it is not implemented in the compiler.

40 A break statement was detected that was not within the scope of a while, do, for, or switch statement. This may occur due to an error in a preceding statement.

41 A case prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

42 The expression defining a case value did not evaluate to an int constant.

43 A case prefix was encountered which defined a constant value already used in a previous case prefix within the same switch statement.

44 A continue statement was detected that was not within the scope of a while, do, or for loop. This may occur due to an error in a preceding statement.

45 A default prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

46 A default prefix was encountered within the scope of a switch statement in which a preceding default prefix had already been encountered.

47 Following the body of a do statement, the while clause was expected but not found. This may occur due to an error within the body of the do statement.

48 The expression defining the looping condition in a while or do loop was null (not present). Indefinite loops must supply the constant 1, if that is what is intended.

49 An else keyword was detected that was not within the scope of a preceding if statement. This may occur due to an error in a preceding statement.

50 A statement label following the goto keyword was expected but not found.

51 The indicated identifier, which appeared in a goto statement as a statement label, was already defined as a variable within the scope of the current function.

52 The expression following the if keyword was null (not present).

53 The expression following the return keyword could not be legally converted to the type of the value returned by the function.

54 The expression defining the value for a switch statement did not define an int value or a value that could be legally converted to int.

55 (W) The statement defining the body of a switch statement did not contain at least one case prefix.

56 The compiler expected but did not find a colon (:). This error message may be generated if a case expression was improperly specified, or if the colon was simply omitted following a label or prefix to a statement.

57 The compiler expected but did not find a semi-colon (;). This error generally means that the compiler completed the processing of an expression but did not find a statement terminator. This may occur if too many closing parentheses were included or if an expression was otherwise incorrectly formed. Because the compiler scans through white space to look for the semi-colon, the line number for this error message may be beyond the actual line where a semi-colon was needed.

58 A parenthesis required by the syntax of the current statement was expected but was not found (as in a while or for loop). This may occur if the enclosed expression was incorrectly specified, causing the compiler to end the expression early.

59 In processing declarations, the compiler encountered a storage class invalid for that declaration context (such as auto or register for external objects). This may occur if, due to preceding errors, the compiler began processing portions of the body of a function as if they were external definitions.

60 The types of the aggregates involved in an assignment or conditional operation were not exactly the same. This error may also be generated for enum objects.

61 The indicated structure or union tag was not previously defined; that is, the members of the aggregate were unknown. Note that a reference to an undefined tag is permitted if the object being declared is a pointer, but not if it is an actual instance of an aggregate. This message may be issued as a warning after the entire source file has been processed if a pointer was declared with a tag that was never defined.

62 A structure or union tag has been detected in the opposite usage from which it was originally declared (i.e., a tag originally applied to a struct has appeared on an aggregate with the union specifier). The Lattice compiler defines only one class of identifiers for both structure and union tags.

63 The indicated identifier has been declared more than once within the same scope. This error may be generated due to a preceding error, but is generally the result of improper declarations.

64 A declaration of the members of a structure or union did not contain at least one member name.

65 An attempt was made to define a function body when the compiler was not processing external definitions. This may occur if a preceding error caused the compiler to "get out of phase" with respect to declarations in the source file.

66 The expression defining the size of a subscript in an array declaration did not evaluate to a positive int constant. This may

also occur if a zero length was specified for an inner (i.e., not the leftmost) subscript of an array object.

67 A declaration specified an illegal object as defined by this version of C. Illegal objects include functions which return arrays and arrays of functions.

68 A structure or union declaration included an object declared as a function. This is illegal, although an aggregate may contain a pointer to a function.

69 The structure or union whose declaration was just processed contains an instance of itself, which is illegal. This may be generated if the * is forgotten on a structure pointer declaration, or if (due to some intertwining of structure definitions) the structure actually contains an instance of itself.

70 The formal parameter of a function was declared illegally as a function.

71 A variable was declared before the opening brace of a function, but it did not appear in the list of formal names enclosed in parentheses following the function name.

72 An external item has been declared with attributes which conflict with a previous declaration. This may occur if a function was used earlier, as an implicit **int** function, and was then declared as returning some other kind of value. Functions which return a type other than **int** must be declared before they are used so that the compiler is aware of the type of the function value.

73 In processing the declaration of objects, the compiler expected to find another line of declarations but did not, in fact, find one. This error may be generated if a preceding error caused the compiler to "get out of phase" with respect to declarations.

74 (W) A string constant used as an initializer for a char array defined more characters than the specified array length. Only as many characters as are needed to define the entire array are taken from the first characters of the string constant.

75 An attempt was made to apply the sizeof operator to a bit field, which is illegal.

76 The compiler expected, but did not find, an opening left brace in the current context. This may occur if the opening brace was omitted on a list of initializer expressions for an aggregate.

77 In processing a declaration, the compiler expected to find an identifier which was to be declared. This may occur if the prefixes to an identifier in a declaration (parentheses and asterisks) are improperly specified, or if a sequence of declarations is listed incorrectly.

78 The indicated statement label was referred to in the most recent function in a goto statement, but no definition of the label was found in that function.

79 (W) More than one identifier within the list for an enumeration type had the same value. While this is not technically an error, it is

usually of questionable value.

80 The number of bits specified for a bit field was invalid. Note that the compiler does not accept bit fields which are exactly the length of a machine word (such as 16 on a 16-bit machine); these must be declared as ordinary int or unsigned variables.

81 The current line contains a reference to a pre-processor symbol that is a circular definition.

82 The size of an object exceeded the maximum legal size for objects in its storage class; or, the last object declared caused the total size of declared objects for that storage class to exceed that maximum.

83 (W) An indirect pointer reference (usually a subscripted expression) used an address beyond the size of the object used as a base for the address calculation. This generally occurs when an expression makes reference to an element beyond the end of an array.

84 (W) A **#define** statement was encountered for an already defined symbol. The first definition is "pushed", so that an additional **#undef** statement is needed to undefine the symbol.

85 (W) The expression specifying the value to be returned by a function was not of the same type as the function itself. The value specified is automatically converted to the appropriate type; the warning merely serves as notification of the conversion. The warning can be eliminated by using a cast operator to force the return value to the function type.

This warning is also issued when a return statement with a null expression (i.e., no return value) appears in a function which was not declared void; generation of the warning for this particular context can be disabled using the **-c** option on the **lc.exe** or **lc1.exe** command.

86 (W) The types of the formal parameters declared in the actual definition of a function did not agree with those of a preceding declaration of that function with argument type specifiers.

87 (W) The number of function arguments supplied to a function did not agree with the number of arguments in its declaration using argument type specifiers.

88 (W) The type of a function argument expression did not agree with its corresponding type declared in the list of argument type specifiers for that function. Note that the compiler does not automatically convert the expression to the specified type; it merely issues this warning.

89 (W) The type of a constant expression used as a function argument did not agree with its corresponding type declared in the list of argument type specifiers for that function. In this case, the compiler does convert the expression to the expected type.

90 The type specifier for an argument type in a function declaration was incorrectly formed. Argument type specifiers are formed according to the rules for type names in cast operators or sizeof expressions.

91 One of the operands in an expression was of type void; this is expressly disallowed, since void represents no value.

92 (W) An expression statement did not cause either an assignment or a function call to take place. Such a statement serves no useful purpose, and can be eliminated; usually, this error is generated for incorrectly specified expressions in which an assignment operator was omitted or mistyped.

93 (W) An object with local scope was declared but never referenced within that scope. This warning is provided as a convenience to warn of declarations that may no longer be needed (if, for example, the code in which the variable was used was eliminated but not its declaration). It may also occur if the only use of the object is confined to statements which are not compiled because of conditional compilation directives such as **#ifdef** or **#if.**

94 (W) An auto variable was used in an expression without having been previously initialized by an assignment statement or appearing in a function argument list with a preceding & (i.e., its address passed to a function). Note that the compiler considers the variable initialized once any statement causes it to be initialized, even though control may not flow from that statement to other subsequent uses of the variable. Note also that this warning will be issued if the third expression in a for statement uses a variable which has not yet been initialized, which may be incorrect if that variable is initialized inside the body of the for statement.

4. INTERNAL ERRORS

These errors are reported via the message:

```
CXERR: xx
```

where **xx** is the error number. When such a message occurs, compilation is terminated immediately, and both the quad file and the object file are probably unusable.

1	Invalid error or warning message code number.
2	The compiler has internally called a function that is not applicable to the current host-target environment.
3	Invalid symbol table access.
4	Declaration chain is broken.
5	An unlink error occurred while processing an "undef" command.
6	The compiler attempted to push back too many tokens.
7	There is no aggregate list for a structure reference.
8	Stack underflow has occurred.
9	Invalid attempt to generate the address of a constant.
10	A test value is not a constant.
11	Invalid unary operator.

12	Invalid binary operator.
13	A scaling object is not a pointer or array.
14	An unexpected end-of-chain occurred while restoring internal context.
15	Invalid quad type.
16	Deletion length is less than two bytes.
17	Insufficient memory.
18	An error occurred when releasing memory.
19	Invalid condition during temporary assignment.
20	Invalid condition while processing program section.
21	Literal pool generation error.
22	Invalid condition while processing data section.
23	Invalid quad file.
24	End-of-file while processing "for" quad.
25	Invalid register number.
26	Temporary save or restore error.
27	Invalid operand size.
28	Invalid storage base.

29	Error during branch folding.
30	Error during control statement processing.
31	Error during special addressing setup.
32	Invalid object description block offset.
33	Too many function parameters.
34	Indirect argument for call-by-reference.
35	Invalid external relocation value.

APPENDIX 4
RECENT CHANGES

This appendix lists the differences between this release and earlier releases of the Lattice AmigaDOS C Compiler.

TABLE OF CONTENTS

1. PURPOSE

This appendix lists the differences between Version 3.03 and Version 3.10 of the Lattice AmigaDOS C Compiler. As always, we've tried to maintain compatibility at both the source level and the object code level. Nonetheless, we could not avoid the following incompatibilities:

1. The Lattice run-time support libraries as distributed by Lattice are not compatible with the Amiga linker **alink** from Metacomco. If, for some reason, you need to use **alink,** you must contact Lattice to obtain copies of the libraries that are compatible with **alink.**

2. You must re-compile any program using floating point because the run-time support library was changed. If you link an old object module with the new libraries, pointers may not be preserved across floating point operations.

3. You must re-compile any programs that pass structures as function arguments, and you must also re-compile any programs that expect structures as function arguments. You do not need to re-compile programs that pass or expect structure pointers as arguments.

4. If you re-compile a module with the **-b** option, you must either use the **-y** option, or re-compile all programs that directly or indirectly call functions in that module. Programs compiled with the **-b** option expect the data section base address to be in A6. Programs compiled previous to this release will not properly propogate this value.

5. You must re-compile any program using the **feof, ferror,** or **fileno** macros from **stdio.h.**

6. Several internal tables and functions have changed. Most users should not be accessing these directly, since they are not described in the normal documentation and we have never guaranteed to preserve them from one release to another. However, if you obtained the Version 3.03 library source code and figured out how to use these items, you should now obtain the Version 3.10 library source and check that the items you are using have not changed. In particular, note that the **_pfmt** and **_sfmt** functions have changed, as have the **UFB** and **_iobuf** structures.

7. If your program was dipping into the Level 2 and Level 1 I/O data structures to obtain such things as AmigaDOS file handles, be advised that this area of the library has changed considerably. In particular, the **open** function now returns an AmigaDOS file handle, so you should not need to examine the **UFB** structure in order to obtain that information.

8. Because of the changes in the Level 1 I/O, it is no longer possible to use file descriptors 0, 1, or 2 to refer to **stdin, stdout,** or **stderr.**

9. Handling of the break characters CTRL-C and CTRL-D has changed since the last release. Previously, the function **Chk_Abort** could be called to determine if a break character had been pressed. Processing of these characters is now controlled by the **onbreak** function, and the UNIX **signal** function. The function **chkabort** may be called to force checking for these characters, but it does not return any values. Normally checking is performed during I/O calls.

2. SUMMARY OF COMPILER CHANGES

1. The standard header files have been modified to include argument types in their external declarations. A new header file, STDLIB.H, contains declarations for functions not covered in the other standard headers.

2. The compiler now consults the AmigaDOS logical name **INCLUDE:** to locate files mentioned on **#include** lines.

   If the file name is enclosed in quotes "...", then the search starts in the current directory, proceeds through directories mentioned on **-i** options, and finishes in the directory associated with the **INCLUDE** logical name. If the file name is enclosed in angle brackets <...>, then the search is restricted to the **INCLUDE** directory.

3. An additional compatability code, **t** has been added to the **-c** option on **lc** or **lc1**. This code disables warning messages for structure and union tags that are used without being defined. For example,

   ```
   struct XYZ *p;
   ```

   would normally produce a warning message if the structure tag XYZ were not defined.

4. **lc** and **lc1** now recognize a **-e** option to handle the extended character sets used in Asian applications.

   Certain Asian characters are represented by two consecutive bytes, with the first byte having its high bit set (i.e. having a value above 127). With the **-e** option, these two-character sequences are recognized and treated as a unit in string and

character constants.

If you code this option as **-e** or **-e0**, then Japanese coding is assumed. This means that all high-bit characters from 0x81 to 0xFC are treated as the start of a two-byte sequence except for those with values from 0xA0 to 0xDF, which are one-byte Katakana codes.

If you code the option as **-e1**, then Chinese and Taiwanese coding is assumed. In other words, no special check is made for the Katakana characters, and so any high-bit character from 0x81 to 0xFC is first half of a two-byte code.

If you code the option as **-e2**, then Korean coding is assumed, and any high-bit character from 0x81 to 0xFD is the first half of a two-byte code.

5. Several of the compile-time limits have been increased, as follows:

LIMIT	**OLD**	**NEW**
Input line size	256 bytes	512 bytes
Substitution text size	256 bytes	512 bytes
Macro arguments	8 args	16 args

Because of these size increases, some very large source programs may cause the compiler to abort with an "Out Of Memory" message.

6. The Amiga Object File Format includes a means of specifying which type of memory each program segment was to be loaded into. The Lattice AmigaDOS C Compiler now supports this option directly. There are two command line flags used with phase 2 of the compiler to specify which segments are to be loaded into specific memory:

```
-cc|d|b         chip memory code|data|bss
-hc|d|b         high speed memory
```

The **-c** flag specifies that the sections designated are to be loaded ONLY in chip ram (the memory accessible to the blitter chip). This memory is generally associated with screen images. Because of a conflict with the compatability flag on phase 1, when using the **lc** driver program this option is referenced via the **-a** flag.

The **-h** flag specifies that the sections designated are to be loaded ONLY in high-speed ram (ram attached to the expansion slot on the right side of the Amiga). Code and data fetches from this memory generally are faster because there is no bus contention with screen dma operations.

Any or all of the segments may be specified with each flag. For instance, the following causes all code to be loaded into high-speed ram and all data to be loaded into chip ram:

```
-cc -hdb
```

The compiler does not check to determine if you have previously specified the segments to be loaded into different ram. It will simply take the specification that occurs last.

If no memory type specification is made then memory type will be determined by availability at load time, with preference given to high speed ram. Be aware that program segments compiled with the **-h** flag will be unable to load on Amiga systems that have not been enhanced by additional memory on the side expansion or custom M68020 boards.

7. The Amiga Load Module Format allows for each program segment to be loaded into separate memory locations. This effect is known as **scatter-loading,** and is very efficient at utilizing fragmented memory. However, this can cause program loading to be excessively time consuming. The Amiga linker has the capability to merge similarly named segments. While this may reduce the efficiency of the memory utilization, it can sometimes significantly improve load times.

   The **-s** command line option for Phase 2 of the compiler is used to control the names of the segments. Used by itself, it causes segments to be named with the default names of **code, data,** and **udata.** It may also be followed with a section name specification, allowing each segment to be individually named. The formats for these specifications are:

   ```
   c=codename d=dataname b=bssname
   ```

   where **codename** is the name to be used with the code section, **dataname** is the name to be used for the initialized data section, and **bssname** is the name to be used for the uninitialized data section. An example command line might contain the following options to name all three segments:

   ```
   -sc=codename -sd=dataname -sb=bssname
   ```

8. The previous release of the Lattice AmigaDOS C Compiler had two debugging options available **(-d),** one for each phase. The **-d** option for Phase 1 caused line number information to be included in the object file. This was primarily used with **omd** to mix source code with the disassembled listing. The **-d** option for Phase 2 caused symbol hunks to be

included in the object file. This symbol information could then be used by symbolic debuggers.

There is now a single debug flag:

-d

Used with Phase 1, it causes symbol hunks to be included in the object file for debugger support, and debug hunks containing line number information to be included for **omd** support.

Because of the amount of extra information in the files, use of the **-d** option can cause object and load module files to be significantly larger than they would be without it. Once programs have been debugged they should be recompiled without the **-d** option to reduce their size.

9. The compiler now supports the Motorola Fast Floating Point **(FFP)** format directly. This means that floating point operations compiled with this option will automatically use the Amiga supplied routines to do the operations in fast floating point format. No source conversion is necessary. This option is used in the command line for Phase 1 of the compiler, and is:

   -f

   FFP is a unique implementation of floating point specifically for the 68000 series, and is not compatible with the **IEEE** format. It is a 32 bit floating point representation that does not support double precision. Use of the **-f** flag with Phase 1 causes the compiler to generate 32 bit FFP storage types and constants.

   It is **not** possible to mix **FFP** and **IEEE** format

operations. The support implemented by the Lattice compiler requires the program to be linked with **ONLY** the appropriate math library. If the **-f** flag is used, the library **lcmffp.lib** must be included before **lc.lib** in the linker command. If the flag is ommitted the library **lcm.lib** must be included before **lc.lib.**

Library opening and closing is performed automatically when needed. The libraries are not opened until an operation which requires the library is performed. This may cause a slight pause during the first execution of a floating point operation (and the first FFP format transcedental function) which may affect benchmark timings.

Note that there is a limited set of transcendental functions available for the **FFP** format. Currently the library only supports those routines supplied by Commodore/Amiga at the time of this release.

10. The **-b** flag has been added to **lc** and **lc1**. This flag directs the compiler to generate data references as offsets from a base address register. This is referred to as **base relative addressing.** This change involved extending the Amiga Object File Format with the addition of new relocation types, and forcing the names of the data and bss hunks to be **__MERGED.** This special name is recognized by **blink** and causes these hunks to be coalesced into a special data section addressed by register A6. This option overrides the use of **-s.** Because of the non-standard relocation types involved, modules compiled with the **-b** flag are not compatible with the Amiga linker **alink.**

    There are potential problems when using this option with programs that will be using the **AddTask** function or may be used as an interrupt handler.

Programs compiled with the **-b** option will be expecting the data section base address to be in address register A6. However, interrupt handlers and tasks created with **Addtask** will potentially have random values in the registers. With these types of programs you **MUST** use the **-y** option for phase 2 that causes register A6 to be loaded with the base address. See the **lc** command in Section C for a complete description.

It is possible to mix modules using base relative addressing with modules that do not. Those routines compiled with the **-b** flag will have their data merged into the special A6-addressable section, while those that were not will continue to be scatter-loaded.

This option is not compatible with the **-c** option for phase 2. If this option is used for either data or bss, it will override the **-b** option.

11. Programs that pass structures as function arguments now copy the entire structure onto the stack before calling the function. Previously a pointer was passed to the called function, which then used the pointer to produce a local copy of the structure.

3. SUMMARY OF LIBRARY CHANGES

1. The translated mode for Level 2 I/O (fopen, fclose, etc.) has been improved so that file positioning works exactly as in non-translated mode. Previously it was not possible in translated mode to obtain a file position via **ftell** and then reposition to the same spot via **fseek**. This improvement required a slight change in **stdio.h**. The change is invisible

to programs that do not use the **feof**, **ferror**, and **fileno** macros and do not directly reference the **_iobuf** structure.

2. The **open** function now returns an AmigaDOS file handle instead of the UFB array index.

3. Earlier versions of the library included a set of I/O functions that interfaced directly with AmigaDOS using file handles. These "level Ø I/O functions" were given peculiar names and were only used internally within the library. We've now made the level Ø functions available under the names **dcreat**, **dcreatx**, **dopen**, **dclose**, **dread**, **dwrite**, and **dseek**. These functions can be used to bypass the UNIX-compatible I/O services that we provide at levels 1 and 2. However, if you are already accessing a file via Level 1 or Level 2 I/O, do not use these Level Ø functions because the higher level services may become confused. This warning applies only to the functions mentioned immediately above. The other file handle functions mentioned below can generally be used in conjunction with Level 1 and Level 2 I/O.

4. Some new I/O functions have been added to provide easy access to AmigaDOS file features. The **getfa** function allows you to get some AmigaDOS file attributes. Similarly, **getft** gets the date/time value stored in the file's directory entry. The **getcd** and **getcwd** functions get the current directory path. See the Index of External Names for additional AmigaDOS functions.

5. Several new functions have been added to provide a more exact simulation of the UNIX and XENIX environments. These include: **abort**, **access**, **asctime**, **assert**, **chdir**, **chmod**, **ctime**, **fdopen**, **getcwd**, **gmtime**, **localtime**, **memccpy**, **memchr**, **memcmp**, **memcpy**,

memset, **mkdir**, **perror**, **qsort**, **rmdir** and **tzset**. See the Index of External Names for additional UNIX/XENIX functions.

6. The **printf** function has been greatly enhanced, with the addition of several format specifications and the elimination of the 200 character output string length limitation.

7. Under AmigaDOS, **argv** will either contain a pointer to an array of argument string pointers, or a pointer to the WorkBench startup message. See the function description for **main** in the External Name Reference section for a complete description.

8. Several library functions have been renamed in order to make them consistent with the rest of the library. These are:

OLD NAME	**NEW NAME**	**DESCRIPTION**
mkext	strmfe	Make file name with extension
mkname	stcgfn	Get node part of file name

9. Unlike the previous version, control-c handling is automatically enabled in programs produced with this release. To disable control-c, you must modify **_main.c** or provide your own handler via the functions **onbreak** or **signal**.

10. In general, the library routines are generated WITHOUT stack overflow checking. However, because of the enormous stack overhead required by AmigaDOS, there is a separate I/O interface object module provided that will check to determine if the required 1K of stack space is available before calling AmigaDOS. This is not part of the standard library because stack checking does not work correctly if a portion of the program is set up as a

separate task using the ROM Kernel **AddTask** function. If the program will not be performing this type of multitasking, however, and file I/O will be required, it is highly recommended that this file (IOSØ.O) be included with the other object modules when linking.

11. Because a task running on the Amiga may not necessarily have an AmigaDOS console window attached to it, the default handlers for stack overflow and control-c do not attempt to output a message to the console. These routines now use an Intuition requester to notify the user of the exception and prompt for the appropriate response. The control-c handler allows the user to continue execution if desired. The stack overflow handler resets the stack to a known state, then waits for an acknowledgement before exiting.

    Both handlers attempt to exit cleanly, closing any open files and deallocating any memory allocated during program execution. This is only effective on files opened with Level 1 or Level 2 I/O function calls, and memory allocated with the Lattice run-time library memory allocation routines. The library now includes the function **onexit** that can be used to establish a function that will be called when the program terminates. If the program will be opening files or allocating memory via direct system calls then a cleanup function should be established using **onexit** to properly handle these system resources.

12. Because of the addition of Fast Floating Point support, the math functions have been separated from the rest of the run-time library. This means there are now three Lattice libraries:

lc.lib I/O, memory alloc., string functions, etc.

lcm.lib IEEE math functions

lcmffp.lib Motorola FFP math functions.

If no floating point operations are involved in the program, then **lc.lib** and **amiga.lib** are the only libraries required. However, if the program uses floating point operations (even just **printf** of a floating point value), then one of the math libraries MUST be included in the linker's library specification BEFORE the other two libraries. Which library to use depends on whether the ffp option **-f** was used. It is NOT possible to use both libraries. You cannot mix **IEEE** and **FFP** formats in a program. Note that this means all object files in a load module, not just all functions in a source file.

To link a program which uses no floating point:

```
blink FROM c.o,file.o TO file LIB lc.lib,amiga.lib
```

To link a program which uses **IEEE** floating point:

```
blink FROM c.o,file.o TO file LIB lcm.lib,lc.lib,amiga.lib
```

To link a program which uses **FFP** floating point:

```
blink FROM c.o,file.o TO file LIB lcmffp.lib,lc.lib,amiga.lib
```

13. The **IEEE** floating point and integer math routines have been optimized, providing an improvement in speed by a factor of 5 to 10.

14. The Amiga library base vectors **DOSBase, Intuition-Base, MathBase, MathTransBase, DiskfontBase,** and

GfxBase are now part of the library. They may be safely referenced as externals in order to avoid possible duplicate declarations by unrelated functions.

4. MISCELLANEOUS CHANGES

1. This release of the AmigaDOS C compiler as distributed by Lattice includes additional utilities that were not previously part of the compiler package.

   oml an object module librarian.

   asm an Amiga compatible 68000 macro assembler.

   blink a public domain Amiga compatible linker.

2. Also, since the libraries distributed by Lattice are not compatible with **alink,** and the compiler is capable of producing files incompatible with **alink,** that linker is no longer distributed by Lattice as part of the compiler package. Lattice supported the development of **blink** by The Software Distillery as a replacement for **alink,** and continues to support The Software Distillery's maintenance and enhancement efforts.

3. It is now possible to invoke the compiler, librarian, and linker with a single command. See the description of the **lc** command in Section C for a complete description.

APPENDIX 5
ASSEMBLY LANGUAGE SYNTAX

This appendix describes the assembly language source file format for the Lattice AmigaDOS 68000 macro assembler.

TABLE OF CONTENTS

ASSEMBLY LANGUAGE SYNTAX

This appendix describes the asembly language source file format for the Lattice AmigaDOS 68000 macro assembler. Execution of the assembler is described in Section C, and the interface between C and assembly language is discussed in the General Information Section 5.3.

The Lattice 68000 AmigaDos assembler conforms closely to the language described in the following Motorola manuals:

1. M68000 Resident Structured Assembler Reference Manual M68KMASM(D4)

2. M68000 16/32-bit Microprocessor Programmer's Reference Manual M68000UM(AD4).

A brief summary of the language syntax is presented in the following sections; exceptions to the Motorola standard are noted, where necessary.

1. SOURCE FORMAT

The general format of an assembly language source line is as follows:

```
label[:] operation operands comment
```

where **label** is an optional label, which if present must begin in the first character position of the line or end with a colon; **operation** specifies either an instruction mnemonic, an assembly directive, or a macro name; **operands** is an expression or list of expressions whose meaning depends on the exact operation which was specifed; and **comment** is any text separated from the preceding operand field by one or more white space

characters. Lines which begin with the character * or which consist entirely of white space are ignored.

Two types of symbols are recognized by the assembler: identifiers and operation symbols. Identifiers are used as labels, and can appear in operand expressions. They are constructed according to the same rules as C identifiers, except that they may be up to 31 characters long, and the period **(.)** is considered an alphabetic character; the dollar sign **($)** may also be used as an embedded (i.e., not the first) character. Operation symbols use the same set of characters, but may be a maximum of 9 characters in length. Note that alphabetic case is significant for identifiers used as labels (i.e., **ABC** is not the same as **abc**), but operation symbols are forced to upper case (i.e., **XQD** is equivalent to **xqd** when used as an operation symbol). The two classes of symbols do not conflict with each other; thus, the same symbol may be used for both a label and an opcode without error.

Constants are specified according to the standard Motorola convention. A decimal constant is formed from a string of decimal digits. An octal constant may be specified by the character **@** followed by octal digits, a hexadecimal constant by the character **$** followed by hex digits, and a binary constant by the character **%** followed by binary digits. Character constants are enclosed in apostrophes, and are limited to a maximum of 4 characters except when used as an operand for **DC.B.** The value is not left-justified or zero-filled if less than 4 characters. An apostrophe can be included in the constant by specifying two successive apostrophes.

The assembler uses a syntax for expressions which recognizes the following operators, listed below with their relative precedence:

(1) - (unary minus)
(2) >> (right shift), << (left shift)
(3) & (bitwise AND), ! (bitwise OR)
(4) * (multiply), / (divide)
(5) + (add), - (subtract)

One or more expressions may appear in the operand string for an assembly directive or a machine operation. An expression, in general, represents a 32-bit value. An absolute expression is one that evaluates to a constant whose value is invariant with respect to any subsequent relocation or linkage of the object module, while a relocatable expression involves an address calculation whose value may depend on the result of relocation or linkage performed on the object module when it is combined with other modules during program linking. Only certain combinations of addition and subtraction operations are permitted on relocatable expression values, as follows:

1. an absolute value may be added to or subtracted from a relocatable value.

2. two values which are relocatable from the same section base or external symbol may be subtracted (yielding an absolute value).

More complex relocatable expressions are not allowed.

2. ASSEMBLY DIRECTIVES AND MNEMONICS

All of the standard 68000 instructions are supported, as described in the Motorola manual. Assembly directives are discussed in the following paragraphs.

Instead of the **section** directive, the Lattice assembler uses the **CSECT** directive of the form:

```
    CSECT     name
```

where **name** is the control section name. Note that a **CSECT** directive must precede any data-defining statements (i.e., there is no default control section). The Lattice 68000 macro assembler uses the concept of named control sections to control the location of data and text during linking; the **CSECT** directive defines the start of a relocatable control section of the specified name. Note that "text" and "data" are used for compatibility with object files generated by the C compiler. Only one instance of a **CSECT** directive with a particular name is permitted; that is, scattered definitions of data for a **CSECT** are not allowed.

The **CSECT** directive may be used to specify additional information about the control section; its general format is:

```
    CSECT     name,type,align,rtype,rsize
```

Only the **name** parameter must be present; it specifies the name of the control section. **type** is zero for code sections, one for data sections , and 2 for uninitialized data sections (like the **udata** section used by the C compiler); the default value is zero (code). **align** specifies the alignment requirements of the control section as a power of two; thus, **align** equal to one means the section will begin on an even address, and a value of two means an address divisible by 4, etc. This parameter is non-functional on the Amiga. All sections are long-word aligned.

The last two parameters specify the type and size of relocation information associated with the address of any data in the control section. **rtype** specifies the relocation type, which determines the meaning of the address, or relocation data; the default value is zero. **rsize** specifies the size, in bytes, of the relocation

data for the section; the default value is four. The legal types and sizes of relocation information on the 68000 are summarized in the following table:

Type	Size (bytes)	Description
0	2 or 4	Absolute short or long address
1	2	PC-relative offset
2	2	Address-register-relative offset

Type 0 relocation results in an absolute address, while type 1 relocation is computed as the difference between the referenced address and the location of the reference. Type 2 relocation is used to assign offsets for address-register-relative addressing of data. Both the base section used for such addressing and any bias placed in the address register are used when a load module is constructed; see Section 5.4.1.

Note that symbols declared in **XREF** statements within a control section will be addressed using the relocation type and size defined for that section. The use of **CSECT** directives which are compatible with the **-b** and **-r** options of the C compiler are described later.

The following additional assembly directives are recognized and supported as in the Motorola assembler:

DS define storage.

DC define constant.

ENDC end of range for IF directive.

EQU defines an equivalence for a symbol.

IFEQ assemble next statements if expression is zero.

IFGE assemble next statements if expression >= 0.

IFGT assemble next statements if expression > 0.

IFLE assemble next statements if expression <= 0.

IFLT assemble next statements if expression < 0.

IFNE assemble next statements if expression is not zero.

INCLUDE read from the named file (useful for macro defs).

OFFSET the base value for set of DS operations defining offsets.

SET defines an equivalence which can be redefined.

XDEF defines one or more externally visible identifiers.

XREF lists external identifiers defined elsewhere.

Note that **IF** directives may be nested, and that an **else** clause can be included by using the **ELSE** directive.

Several directives are tolerated but ignored:

PAGE, TTL, SPC, IDNT

The various structured control statements are not supported, nor are the following directives:

ORG, MASK2, REG, COMLINE, FAIL, OPT, NOOBJ, LLEN, NOPAGE, LIST, NOLIST, FORMAT, NOFORMAT

3. MACROS

The Lattice assembler implements macros in a form quite different from that in the Motorola assembler. Macros are specified according to the following model (brackets enclose optional specifiers):

```
             MACRO
[labelarg]   macroname [arglist]
             .
             .
             .
             ENDM
```

Note that the **MACRO** directive appears by itself, followed by a line which serves as the macro definition model, which is then followed by additional lines of text terminated by an **ENDM** directive. The lines between the model and the **ENDM** directive represent the expansion text of the macro. The model specifies the identifiers which are recognized inside that text, and values from the invocation of the macro are substituted for them.

labelarg, if present, is an identifier which can be used within the expansion text to obtain the label, if any, used on the macro invocation. **arglist** is a comma-separated list of strings of the format

```
argsym[=default]
```

where **argsym** is an identifier symbol which can be used within the expansion text to obtain the corresponding argument text used on the macro invocation line. **default** is an arbitrary string of characters which will be used if the corresponding macro argument is null. Note that **default** cannot contain any white space characters unless they are enclosed in single or double quotes.

A simple example illustrates some of these features:

```
         MACRO
LABEL    COPY    SOURCE,DEST,LEN
LABEL    MOVE.W  #LEN-1,D0
         MOVE.L  #SOURCE,A0
         MOVE.L  #DEST,A1
CPLOOP   SET     *
         MOVE.B  (A0)+,(A1)+
         DBF     D0,CPLOOP
         ENDM
```

This macro might be invoked as

```
         COPY    CTR1,CTR2,100
```

resulting in the expanded text

```
         MOVE.W  #100-1,D0
         MOVE.L  #CTR1,A0
         MOVE.L  #CTR2,A1
CPLOOP   SET     *
         MOVE.B  (A0)+,(A1)+
         DBF     D0,CPLOOP
```

Note that a label argument (such as **LABEL** above) must be specified and placed on the first expansion line of the macro if the ability to use a label on the macro invocation is desired. If expansion of the macro is to be terminated early (inside an **IF** sequence, for instance), the directive **EXITM** can be used.

4. RELATIVE ADDRESSING

The Lattice assembler can be used to generate assembly language modules which are compatible with the conventions used in C programs compiled with the **-b** option on **lc1** or the **-r** option on **lc2**. PC-relative external function calls can be forced using sequences such as the following:

```
            CSECT     text,0,,1,2
            XREF      cfunc
            XDEF      afunc
    afunc   JSR       cfunc(PC)
            RTS
```

This mechanism is the same as that used for function calls in code compiled with the **-r** option on **lc2.** Note that the **XREF** statement for the external function being called must appear after the **CSECT** statement, and that all calls must be made in the PC-relative addressing mode. If the **XREF** statement for the external function precedes the **CSECT** statement a standard JSR to a four-byte external address can be used.

Address-register-relative addressing of data can be forced if the data is placed in a control section such as the following:

```
            CSECT     data,1,,2,2
            XREF      xdata
    data1   DC.W      10
    data2   DC.L      20
```

It then must be addressed using the appropriate address register:

```
            MOVE.W    data1(A5),D0
            MOVE.L    data2(A5),D1
            MOVE.L    D1,xdata(A5)
```

This mechanism is the same as that used for addressing data in code compiled with the **-b** option on **lc1.** Note that the **XREF** statement for any external data elements being addressed must appear after the **CSECT** statement, and that all data references must be made in the address-register-indirect-with-displacement addressing mode. If the **XREF** statement for the external data precedes the **CSECT** statement, a standard reference to a

four-byte external address can be used.

SECTION C
COMMANDS

This section describes the AmigaDOS commands that are provided with the Lattice compiler package or that must be used in conjunction with the compiler.

C O N T E N T S

NAME

asm..................68000 Macro Assembler

SYNOPSIS

```
asm >listfile options filename
```

DESCRIPTION

The AmigaDOS 68000 macro assembler is designed to generate assembly language components for inclusion in C programs, but it can also be used to generate programs entirely written in assembly language. It supports the full set of 68000 mnemonics, as well as an extensive set of assembler directives and a powerful macro facility. The syntax of assembly language statements are described in Appendix 5.

The assembler reads an assembly language source file and produces an object file in the Amiga object file format, along with an optional listing of the source and assembled code. The format of the command to invoke the assembler is:

```
asm [>listfile] [options] filename
```

The various command line specifiers are shown in the order they must appear in the command. Optional specifiers are shown enclosed in brackets.

>listfile

Causes the listing and error message output of the assembler to be directed to the specified file.

options

Assembler options are specified as a hyphen followed by a single letter; in some cases, additional text may be appended. The letter may be supplied in either upper or lower case. Each option must be specified separately, with a separate hyphen and letter (that is, they cannot be combined as they can for certain UNIX programs). Current options include:

-c

This option specifies that the sections designated by the characters which immediately follow are to be loaded into memory addressable by the Amiga's custom chips. This is necessary for screen image data. The **-c** option must be immediately followed by one or more of the following letters in any order:

c code segment

d data segment

b bss or uninitialized data

The default action is to load the segments into memory not addressable by the custom chips if it is available. This is generally desirable because it avoids bus contention between the processor and the custom chips. Image data, however, must be loaded into memory accessible to the custom chips. For example,

```
-acdb
```

will cause all segments to be loaded into chip addressable memory, regardless of the system memory configuration.

-iprefix

Specifies that INCLUDE files are to be searched for by prepending the filename with the string prefix, unless the filename in the INCLUDE statement is already prefixed by a drive or directory specifier. Up to 4 different **-i** strings may be specified in the same command. When an unprefixed INCLUDE filename is encountered, the current directory is searched first; then file names are constructed and searched for, using prefixes specified in **-i** options, in the same left-to-right order as they were supplied on the command line. No intervening blanks are permitted in the string following the **-i**. Note that if a directory name is to be specified as a prefix, a trailing slash must be supplied.

-l[list]

Causes a listing of the source file to be written to the standard output. The listing displays the appropriate location counter and data generation information alongside the assembly source. One or more of the following characters may be appended to the **-l** option, with the following effects:

x Lists the expansion text for macros.

m Lists additional data generated for source lines which cannot be accommodated alongside the original source

line (i.e., allows multiple listing lines for each source line).

i Lists the source for text from INCLUDE files as well as the original source file.

-oprefix
Specifies that the output filename (the .o file) is to be formed by prepending the input filename (the .a file which is being assembled) with **prefix.** Any drive or directory prefixes originally attached to the input filename are discarded before the new prefix is added. No intervening blanks are permitted in the string following the **-o.** Note that if a directory name is to be specified as a prefix, a trailing slash must be supplied.

filename
Specifies the name of the assembly language source file which is to be assembled; this is the only command line field which must be present. If the filename is specified without an extension, .a is assumed. The object file created by the assembler will have the same name as the source file, except that .o will be supplied in place of any extension.

EXAMPLES

This command causes the assembly language source file

modn.a to be assembled, producing the object file **modn.o**. A listing of the source file, along with any error messages generated, will be written to the file **modn.lst**.

```
asm >modn.lst -l modn
```

logical name assignment **Class: AMIGA**

NAME

assign..............logical name assignment

SYNOPSIS

```
assign logical_name  path
```

DESCRIPTION

The AmigaDOS **assign** command is used to establish a relationship between an arbitrary name and a disk directory or file. Once the assignment of a name to a directory or file is made, you can use the logical name in place of the directory or file specification. For example,

```
assign myfiles: mydisk:project/source
```

relates the logical name **myfiles:** to the directory **project/source** on the disk volume **mydisk:**. After this assignment is made, the file specification

```
myfiles:somefile.c
```

would be synonomous with

```
mydisk:project/source/somefile.c
```

Logical name assignments may be removed by making the assignment without the second parameter. For example,

```
assign mydisk:
```

removes any previous assignment of the logical name **mydisk:**.

logical name assignment **Class: AMIGA**

The Lattice AmigaDOS compiler uses various logical names to help locate files required to compile and link C programs. See section 'E' for more information on these specific logical names.

SEE ALSO

lc

NAME

blink...............Amiga Linker

SYNOPSIS

```
blink FROM obj TO exe LIB lib MAP map   (Form 1)

blink WITH file                         (Form 2)
```

DESCRIPTION

The object modules produced by the Lattice C Compiler are normally combined with the Lattice libraries and the Amiga library to form an executable module, also called a load module. To accomplish this the Lattice AmigaDOS C Compiler includes **blink,** an Amiga format linker produced by The Software Distillery. Other linkers can also be used if they support the Amiga Object File Format. The **-L** option of the **lc** command can be used to invoke **blink. Blink** is fully compatible with the Amiga linker **alink,** accepting all **alink** command line options and WITH files. In addition, **blink** supports many additional options not found in **alink.**

The linker accepts two types of input: object files and library files. There may be more than one of each type. Then it produces a single executable file and, optionally, a load map file.

Blink is driven by keyword parameters in any order indicating the action to be performed. The basic syntax is:

```
blink [FROM][ROOT] files [TO file] [WITH file] [VER file]
[LIBRARY|LIB files] [XREF file] [options] [MAP file map_options]
```

where:

file means a single file.

files means zero or more file names separated by a comma, plus sign or space.

and the following keywords are recognized:

FROM files
specifies the object files that are the primary input to the linker. These object files will always be copied to the root of the object module. You must specify at least one object file for the root. If it appears as the first option to **blink** then the **FROM** keyword is optional and may be omitted. **ROOT** is an acceptable synonym for **FROM**. **FROM** may be used more than once with the files for each **FROM** adding to the list of files to be linked.

TO file
specifies the target executable module to create. If omitted it defaults to the same name as the first object module listed on a **FROM** option with its .o suffix removed. Note that because the first object module must be the startup routine **c.o,** if the **TO** option is omitted the load module will be called **c.**

WITH file
specifies a file containing **blink** command options to be processed for this link. More than one **WITH** file may be specified, and **WITH** files may contain additional **WITH** statements. The contents of all **WITH** files will be treated

as if they were specified on the **blink** command line.

VER file
specifies a destination file to contain all linker output messages. If you do not specify this keyword then all messages go to the terminal.

LIBRARY files
specifies the files to be scanned as libraries. Only referenced segments from the library files will be included in the final object module. **LIB** is a valid synonym for **LIBRARY.**

XREF file
specifies that a cross reference listing is to be produced. The **file** parameter is accepted for compatability with **alink,** but the name is ignored. The cross reference listing is written to the map file.

FASTER
is a do-nothing option that is included only for **alink** compatibility.

VERBOSE
causes **blink** to print out the names of each file as it processes it.

NODEBUG
supresses any HUNK_DEBUG, symbol table information, or hunk names in the final object file. This is equivalent to the object file that would be produced if the Amiga utility **stripa** were run on the final object file.

SMALLDATA

causes all DATA and BSS sections to be coalesced into a single hunk. This is necessary for linking C programs that were not compiled with the **-b** option with programs that reference data in those modules using base register relative addressing. It can also be used to compact the data hunks into a single hunk. This results in a smaller executable load module, but can make it difficult to load the program if system memory is very fragmented.

SMALLCODE

causes all code sections to be coalesced into a single hunk. This is useful when programs compiled with the **-r** option call functions that would not normally be coalesced into the same hunk.

WIDTH n

sets the maximum line length for the map and cross reference listings. This is useful when sending the output to a device that has different line length requirements. If not specified it defaults to 8Ø.

OVERLAY

specifies the start of an overlay tree terminated by a line consisting of a single pound sign '#'.

MAP file options

specifies a file to which a map is to be written. The options control which parts of the map will be written. For more information on the **MAP** option see the description of the map options which follows.

Blink Map Options

Blink is capable of producing reports in a variety of formats. The basic format of the **MAP** keyword is

```
MAP [[filename],option,option,...]
```

where

filename is the map output file

option can be either a letter designating which report to produce or additional keywords that provide formatting information.

The linker is capable of writing many different kinds of reports to the map file. Each report is designated by a single character, and any or all of the reports may be generated.

f Files. This report lists each file that contributes to the load module, the program unit name of each program included from the file, and the name, size, type, and location of each hunk from that program unit.

h Hunks. This is the default if no options are specified. It lists the hunk number, name, type, and total size, and the file name, program unit name, and beginning offset of each file contributing to that hunk.

l Libraries. This report is the same as for files, but includes information for hunks pulled in from the libraries.

- **o** Overlays. This is a report of the overlays generated by the linker.

- **s** Symbols. This report lists the hunk number and offset of every external symbol defined in the load module.

- **x** Xref. This is a listing of each symbol, its location, and the location of every reference to that symbol.

Each page will have up to three title lines, the last of which is optional. The first line will give the program name, section name, and page number. The second line is blank. The third line is used for a column header if needed. Additional formatting information can be provided by specifying one or more of the following keywords:

WIDTH n This specifies the number of columns allowed in the map file. The default width is 80 columns.

INDENT n This specifies the number of columns to indent on a line. This amount is included in the total width specified with the WIDTH keyword. The default is 0.

HEIGHT n This is the total number of lines on a page in the map file. A value of 0 indicates no pagination. The default value is 55.

HWIDTH n This keyword option specifies the width of hunk names. The default is 8.

FWIDTH n This specifies the width of file names. The default is 16.

PWIDTH n This specifies the width of program unit names. The default is 8.

SWIDTH n This specifies the width of symbol names. The default is 8.

FANCY This keyword option instructs the linker to write printer control characters to the map file. This is the default condition.

PLAIN This option turns off the FANCY option.

If the FANCY option is on, titles and headers will be underlined and the output will be separated into pages with form feed characters. The page number is given in two parts: the section number and the page number within that section.

EXAMPLES

```
blink FROM LIB:c.o+myprog.o TO myprog LIB LIB:lc.lib+LIB:amiga.lib
MAP myprog.map,h,s,f,o,l
```

links the object file named **myprog.o** with the Lattice startup routine **c.o** and with the Lattice and Amiga standard libraries, named **lc.lib** and **amiga.lib** respectively. The executable file is called **myprog**, and the map file is called **myprog.map**.

NAME

lc..................Lattice C Compiler

SYNOPSIS

```
assign LC: executable path
assign INCLUDE: include path
assign LIB: library path
assign QUAD: quad path

lc options files
```

DESCRIPTION

The normal method for executing the Lattice C Compiler is to invoke the **lc** command. This program separates the **options** list into those for pass 1 and those for pass 2. Then it executes **lc1** (pass 1) and **lc2** (pass 2) for each of the C source files specified by the **files** list. This list can consist of one or more file names and/or file patterns, separated by white space. For example,

```
lc #? :mydir/myprog :mydir/abc?
```

will compile all C source files in the current directory, plus the source file named **:mydir/myprog.c,** plus all C source programs in the **:mydir** directory that have four-character names beginning with "abc". Note that the **lc** command automatically supplies the .C extension on all source file names.

The LC command will also automatically invoke the librarian and linker if required.

This command utilizes the logical name assignment feature of AmigaDOS to locate the various programs and

files. These assignments allow these programs and files to be located in any directory on any disk. Refer to Section 'E' for a complete description of these logical names.

The **options** list can contain any combination of the following, separated by blanks:

-a This option specifies that the sections designated by the characters which immediately follow are to be loaded into memory addressable by the Amiga's custom chips. This is necessary for screen image data. This option is synonymous with the **-c** option for **lc2**. The **-a** option must be immediately followed by one or more of the following letters in any order:

c code segment

d data segment

b bss or uninitialized data

The default action is to load the segments into memory not addressable by the custom chips if it is available. This is generally desirable because it avoids bus contention between the processor and the custom chips. Image data, however, must be loaded into memory accessible to the custom chips. For example,

```
-acdb
```

will cause all segments to be loaded into chip addressable memory, regardless of the system memory configuration.

-b This option causes the compiler to use a base-relative form of addressing when referring to data section locations. This is a more compact instruction format that uses a 16-bit offset field instead of a 32-bit absolute address. The data is addressed relative to the contents of address register A6. This method has the disadvantage of only allowing 64K bytes of data that can be addressed using this option, and all the data addressed this way must be in the same hunk. The default is to use 32-bit absolute addressing, which allows data hunks of virtually unlimited size that can reside anywhere in memory.

This option cannot be used with programs that will be used as interrupt handlers or that will be used with the **AddTask** function unless the **-y** option is also used.

-C This option causes the LC command to continue with the next source file when a fatal compilation error is reported while multiple source files are being compiled. Normally, a fatal error causes the process to pause with the following message displayed on your screen:

```
Compiler return code xx.
Press Y to abort, any other key to continue.
```

The compiler error messages are also displayed immediately above this prompter. Then if you respond with a 'Y', LC will abort; otherwise, it will proceed to the next source file.

-c The compiler defaults to the ANSI C language standard, but the **-c** "compatibility option" can

be used to activate some deviations from the standard, and in that way achieve compatibility with previous versions of Lattice C. The **-c** must be immediately followed by one or more letters from the following list, in any order:

c Allows comments to be nested.

d Allows $ character to be used in identifiers.

m Allows use of multiple character constants (e.g. 'ab').

s Causes compiler to generate a single copy of identical string constants.

t Enables warning messages for structure and union tags that are used without being defined. For example,

```
struct XYZ *p;
```

would not normally produce a warning message if the structure tag XYZ were not defined.

u Forces all "char" declarations to be treated as "unsigned char".

w Shuts off warning messages generated for "return" statements that do not specify a return value within an "int" function. By the ANSI standard, such functions should be declared as "void" instead of "int".

-d This option has two uses. When there is no variable field following the **-d,** it activates the debugging mode. Additional information is placed into the object module to facilitate symbolic debugging. Also, the preprocessor symbol **DEBUG** is defined so any debugging statements in the source file will be compiled.

The **-d** option can also be used to define preprocessor symbols in the following ways:

-dsymbol
Causes **symbol** to be defined as if you had included the statement

```
#define symbol
```

in your source file.

-dsymbol=value
Causes **symbol** to be defined as if you had included this statement in your source file:

```
#define symbol value
```

-e This option causes the compiler to recognize the extended character set used in Asian-language applications.

Certain Asian characters are represented by two consecutive bytes, with the first byte having its high bit set (i.e. having a value above 127). With the **-e** option, a two-byte Asian character is treated as a unit in string and

character constants. This means that when the compiler scans a text string enclosed in double quotes, it will recognize the first byte of an Asian character and suppress lexical analysis of the second byte. So, if the second byte is a backslash or a double quote, it will not receive any special processing.

Also, if you enclose an Asian character in single quotes, the compiler will produce a two-byte constant, and if you have not specified the **-cm** option, it will warn you that multi-character constants are not allowed. In general, you should avoid placing single quotes around an Asian character because the resulting multi-character constant is not portable. On the iAPX86, the first byte will be in the low-order part of the constant, while on the 68000 it will be in the high-order part.

The **-e** option can be specified in the following ways:

-e or -e0

Japanese coding is used. This means that characters with values from 0x81 to 0x9F and 0xE0 to 0xFC are treated as the start of a two-byte sequence. The characters from 0xA0 to 0xDF are single-byte Katakana codes.

-e1

Chinese and Taiwanese coding is used. This means that characters with values from 0x81 to 0xFC are treated as the start of a two-byte sequence.

-e2

Korean character coding is used. This means that characters with values from Øx81 to ØxFD are treated as the start of a two-byte sequence.

-f This option causes the compiler to use the Motorola Fast Floating Point format for all floating point operations. The resulting object module must be linked with the FFP math library **lcmffp.lib.** Normally the compiler uses the IEEE standard for floating point representation, which is compatible with the MC68881 floating point coprocessor. Files compiled with this option are not compatible with files that use the IEEE format, and may not be linked with object modules that use the IEEE format.

-h This option specifies that the sections designated by the characters which immediately follow are to be loaded into memory not addressable by the Amiga's custom chips. The **-h** option must be immediately followed by one or more of the following letters in any order:

c code segment

d data segment

b bss or uninitialized data

The default action is to load the segments into memory not addressable by the custom chips if possible. This option prevents the specified segments from being loaded into chip addressable memory even if no other memory is available.

This can cause programs to not execute if external high-speed memory is not available on the machine. As an example,

```
-hcdb
```

would cause all three sections to be loaded only in high-speed ram.

-i This option specifies a directory that the compiler should search when it is attempting to find an include file that is enclosed in double quotes. For example, if you specify the option as **-idfØ:headers/ -idf1:local/** and then place the line

```
#include "defs.h"
```

in your source program, the compiler first tries to find the header file named **defs.h** in the current directory. If it's not there, then the compiler looks for **dfØ:headers/defs.h** and **df1:local/defs.h** in that order. Finally, if those attempts fail, the compiler continues the search using the directories specified in the INCLUDE logical name assignment.

Note that you can place up to ten **-i** options on the command line.

-L When this option is present, **lc** invokes the linker if all compilations are successful. The first source file name is used as the name of the executable and map files produced by the linker. Any other files that were compiled are supplied to the linker as secondary object files. The Lattice C startup routine is included

as the first object module.

The linker is instructed to search the standard Lattice library file **lc.lib** and the standard Amiga library file **amiga.lib**.

Additional Lattice libraries and linker options may be specified by immediately following the -L option with one or more of the following letters:

c This invokes the SMALLCODE option of the linker. It causes all code hunks to be combined into a single code hunk. **-b** option.

d This invokes the SMALLDATA option of the linker. It causes all data and bss hunks to be combined into the single data hunk that is referenced by modules compiled with the **-b** option.

f This letter specifies that the Motorola FFP math library **lcmffp.lib** is to be searched before the standard run-time support library.

h This letter directs the linker to output the hunk portion of the map. This is the default map if no other map options are specified.

l This letter directs the linker to include library information in the map file.

m This letter specifies that the Lattice IEEE math library **lcm.lib** is to be

searched before the standard run-time support library.

o This directs the linker to include overlay information in the map file.

s This letter directs the linker to produce a symbol listing in the map file.

v This invokes the VERBOSE option of the Software Distillery linker. It causes the linker to display statistical messages as it is processing the object files and libraries.

For example, **-Lm** will search **lcm.lib** before **lc.lib** and **amiga.lib**, and **-Lvf** will search **lcmffp.lib**, **lc.lib**, and **amiga.lib**, and display messages regarding the current linker status. Note that the math library options are mutually exclusive, and that the standard libraries are always searched last.

If you want to search other libraries, you must list those libraries after the letters options (if any), and use plus signs as separators. For example, **-L+myfuncs.lib** searches **myfuncs.lib** before the standard Lattice and Amiga libraries, while **-Lm+myfuncs.lib+:george/myfuncs.lib** searches the libraries **myfuncs.lib**, **:george/myfuncs.lib**, **lcm.lib**, **lc.lib**, and **amiga.lib**. Note that the special libraries are searched before the Lattice libraries.

The **-L** option creates a file in the current directory named **xxx.lnk**, where **xxx** is the name of the first source file to be compiled (i.e.,

the same name that is used for the executable and map files). This .LNK file serves as input to the linker, and it is not deleted at the end of the procedure. This allows you to easily re-link if, during your testing, you find a need to change and re-compile only one module. To do this, simply execute the linker utility **blink** in the following way:

```
blink WITH xxx.lnk
```

where **xxx.lnk** is the name of the .LNK file previously produced by the LC command.

-l This option causes all objects except characters, short integers, and structures that contain only characters and short integers to be aligned on longword boundaries (i.e., addresses evenly divisible by 4). Structures will be longword aligned if they contain any members that must be aligned.

-M When this option is present, LC will compile only those source files with dates more recent than the corresponding object files. Note that the dates of included files are not checked. In other words, if you change a header file without changing the source file that includes it, the source file will not automatically re-compile because it still pre-dates its object file.

-n This option causes the compiler to retain only 8 characters for all identifiers. The default maximum identifier length is 31 characters. In either case, anything beyond the maximum length is ignored.

-o This option should be followed by the drive, directory, or complete file name for the object file that is produced by pass 2. Several examples are:

-odfØ:	Places the object file in the root directory on drive dfØ:.
-o:obj/	Places the object file into directory **obj/** on the current drive. The name of the file is the same as the source file name, with a .O extension instead of .C.
-ospecial.o	Places the object file into the current directory with the name **special.o.**

-q This option should be followed by the drive, directory, or complete file name for the "quad file", which is the intermediate file generated by pass 1 and read by pass 2. Several examples are:

-qram:	Places the quad file in ram disk.
-qdfØ:	Places the quad file in the root directory of drive dfØ:.
-q:quads/	Places the quad file into directory **quad/** on the current drive. The name of the file is the same as the source file name, with a .Q extension

instead of .C.

-qspecial.qqq Places the quad file into the current directory with the name **special.qqq**.

Note that the quad file is automatically deleted after pass 1. If you don't want this to happen, call the **lc1** command directly.

-r This option is used to cause the compiler to use pc-relative addresses instead of absolute addresses for external function calls. Target locations must be within +/-32K of the instruction. For small programs this form of branching can be very efficient. However, in larger programs the target function may not be within the necessary range. If the linker being used does not automatically adjust these instructions then such programs may not be linkable. If the linker does support the automatic adjustments necessary, large programs may run slower because out-of-range branches must be routed through a jump table. In general, however, this option can produce more compact load modules.

-R When this option is specified, the object modules produced by the compiler are automatically inserted into a library file, replacing modules of the same names. The option must be followed by the name of the library, as in

```
-Rmylib.lib
```

which places the object modules into the **mylib.lib** library file. The **-R** can be followed by any valid file name, including drive code and

path. A .lib extension is not automatically supplied.

-s This option allows you to change the default hunk or segment names generated by the compiler. If it is not present, the hunks are unnamed. You can use this option to provide hunk names as follows:

-s
This causes the compiler to use the default names of **text** for the program section, **data** for the data section, and **udata** for the bss or uninitialized data section.

-sc=codename
Causes the compiler to use the name **codename** for the program, or code, section without affecting the names of the other sections.

-sd=dataname
Causes the compiler to use the name **dataname** for the data section without affecting the names of the other sections.

-sb=bssname
Causes the compiler to use the name **bssname** for the bss, or uninitialized data, section without affecting the names of the other sections.

-u This option by itself undefines all preprocessor symbols that are normally pre-defined by the

compiler. If you want to undefine a specific symbol, it must immediately follow the **-u**. For example, **-uAMIGA** undefines the AMIGA symbol.

-v This option disables the generation of stack checking code at the beginning of each function.

-x This option causes all global data declarations to be treated as externals. This can be useful if you define data in a header file that is included by multiple source files. The **-x** option can be used with all the files except one, in this case, to cause the data items to be defined in one module and referenced as externals in the others.

-y This option causes each function entry sequence to load address register A6 with the value of the linker defined symbol **LinkerDB.** This symbol is the data section base address, biased as necessary. This option must be used if the **-b** option is used with interrupt code or functions that will be multi-tasked with the **AddTask** function. Note that in general only the functions that can be used as entry points to the task or interrupt handler need to use this feature, since register A6 will be propagated by subsequent function calls.

RETURNS

The LC command returns the following completion codes:

0 All compilations were successful. That is, at least one source program was compiled, and there were no fatal errors.

1 One or more fatal compilation errors were reported.

2 No source files were found.

SEE ALSO

lc1, lc2, link

EXAMPLES

The first command assigns the INCLUDE logical name so that header files will be found in the **df0:include** directory. The second command assigns the LC logical name so that the driver program can find the executables in the **df0:c** directory. The next command assigns the LIB logical name so that the driver program can tell the linker to locate the standard library files in the **df0:lib** directory. The next command compiles all modules in the **mylib** subdirectory and then constructs a library named **mylib.lib** if all compilations are successful. The last command compiles **myprog.c** and links it with **mylib.lib** to produce a load module named **myprog**.

```
assign INCLUDE: df0:include
assign LC: df0:c
assign LIB: df0:lib
lc  -Rmylib mylib/#?
lc -mp -L+mylib myprog
```

Compiler pass 1 **Class: LATTICE**

NAME

lc1.................Compiler pass 1

SYNOPSIS

```
assign INCLUDE: include path

lc1 options file
```

DESCRIPTION

This command invokes the first compiler pass, which reads a source file and translates it into an intermediate form known as a "quad file". Unlike the **lc** command, you can only specify one source file to **lc1**, and it should be written without the .C extension. For example, if the file argument is **myprog**, this pass will translate **myprog.c** into the quad file named **myprog.q**.

The **options** argument can consist of the following items, which are described in the **lc** command:

-b Base register relative data addressing.

-c Compiler compatibility settings.

-d Debugging mode or preprocessor symbol definition.

-e Extended character set processing.

-f Motorola Fast Floating Point format.

-i Directory paths for local include files.

-l Longword alignment of data items.

-n Long symbol names.

-o Specify destination for .Q file. Note that this option is specified as **-q** on the **lc** command.

-p Preprocess only. This option is not available with the **lc** command. When this option is specified, **lc1** only performs the preprocessor phase and writes the output to a file of the same name with a .p extension.

-u Undefine preprocessor symbols.

All options must appear before the file name.

SEE ALSO

lc, lc2

NAME

lc2.................Compiler pass 2

SYNOPSIS

lc2 options file

DESCRIPTION

This command invokes the second compiler pass, which reads a quad file and translates it into an object file. Note that you can only specify one quad file to **lc2**, and it should be written without the .Q extension. For example, if the file argument is **myprog**, this pass will translate **myprog.q** into **myprog.o**.

The **options** argument can consist of the following items, which are described in the **lc** command:

-c Chip memory specification. This is the same as the **-a** option on the **lc** command.

-h High speed memory specification.

-o Specify destination for .O file.

-r Use pc-relative branches for function calls.

-s Specify segment name.

-v Disable stack checking.

-y Unconditionally load the base register

All options must appear before the file name.

SEE ALSO

lc, lc1

NAME

omd.................Object Module Disassembler

SYNOPSIS

omd >output options object source

DESCRIPTION

This utility program disassembles an object file produced by the Lattice C Compiler and produces an output listing consisting of assembly-language statements, possibly interspersed with the original C source code. The **object** field is required and gives the complete object file name. That is, OMD will not automatically supply the .o extension. The **source** field is optional. If present, it must be the complete source file name. When this field is used, you should have compiled the source file with the **-d** option (see the **lc** command) to produce the debugging information in the object module that allows **omd** to associate a particular source line with the object code that was generated. If you did not use the **-d** option, then the C source lines will not appear in **omd's** output.

The **>output** field is optional and redirects **omd's** output from the screen to an output device or file. Most people use **omd** by redirecting its output to a file and then printing the file. See the example below.

The **options** field need not be present. The Amiga implementation of **omd** only accepts the following option:

-x This option overrides the default size of the buffer used to hold the external symbol section

of the object module. For example, **-x200** establishes a buffer that can hold 200 external symbols, which is the default. You should increase this value if **omd** reports that there are too many external symbols.

EXAMPLES

This example compiles MYPROG.C to produce MYPROG.O, which is then disassembled. The disassembled listing is saved in the file MYPROG.LST.

```
lc -d  myprog
omd >myprog.lst myprog.o myprog.c
```

Object Module Librarian **Class: LATTICE**

NAME

oml..................Object Module Librarian

SYNOPSIS

oml [<cmdfile] [>listfile] [options] libfile [commands]

DESCRIPTION

The object module librarian **oml** can create a library file by combining object modules, generate a listing of the modules (and their public symbols) contained in a library, or manipulate modules within an existing library file.

Library files provide a convenient way of collecting object modules to be presented as a group of available components during program linking; the load module builder then includes only those modules from the library which are actually needed by the load module being built. Libraries are especially useful when several programs make use of common subroutines, since these subroutines can be placed in a library file and included on an "as-needed" basis when the programs are linked.

A library file is made up of object modules, each of which was originally a single file. Each module within the library is identified by a module name, which is normally obtained from the object module itself (the Amiga object module format defines a special "program unit" or module name record). This name is placed in the object module by the translator (assembler or compiler) program which generates it. Some modules may not contain a module name at all; in that case, the object module librarian assigns a module name of the form **$nnn,** where **nnn** is a decimal number indicating

that the module was the **nth** unnamed module encountered in the library.

In order to perform replacement of modules within a library file, it is necessary to insure that the module contains a program unit or module name. The Lattice AmigaDOS C Compiler and AmigaDOS assembler always place a module name in the object files they produce. The current versions of the compiler and assembler use the name of the object file. Thus, compiling **ftoc.c** produces an object module with the name **ftoc.o**.

The Metacomco Amiga assembler does not place any module name in the output file unless the IDNT directive is used. For example:

```
IDNT     "asmod.o"
```

causes the module name **asmod.o** to be placed in the object file created when the source file is assembled.

When the linker examines a library file to find modules to be incorporated into a program, the module name is not important; instead, the linker decides if a module is needed on the basis of the public symbols it defines. A module may define one or more such symbols, which identify program components such as functions or data elements. Because the presence of more than one definition for a symbol may cause confusion, the object module librarian warns when it examines or constructs a library file which includes multiple definitions of a symbol.

Each invocation of **oml** specifies a particular library file upon which operations are to be performed. Then, a sequence of one or more commands is used to indicate

the desired operations.

Commands may be specified on the command line used to execute **oml**, or they may be read from **stdin**, or a combination of both kinds of input may be used. If no commands are specified on the command line which executed **oml**, they are automatically read from **stdin**, which is usually the user's console but can be redirected to a file. The special command @ (valid only on the command line) is used to switch command input from the command line to **stdin**; an explicit file name may be attached to the @ to force commands to be read from that file. Commands are read from a file or from **stdin** until an end of file condition is detected; if commands are being read from the user's console, a control-backslash must be used to end command input.

Each command is specified as a single character, usually followed by a list of module names or object file names. Commands and file/module names are separated from each other by white space. If a command is followed by one or more names, the first name specified is NOT checked as a possible command; thus, names which might be confused with commands must be specified as the FIRST name following a command.

The format of the command to invoke the object module librarian is:

```
oml [<cmdfile] [>listfile] [options] libfile [commands]
```

The various command line specifiers are shown in the order they must appear in the command. Optional specifiers are shown enclosed in brackets.

<cmdfile

Causes commands to be read from the named file,

provided that (1) no commands are specified on the command line or (2) the @ command (see below) is used to force commands to be read from **stdin.** If this option is omitted and neither of the above conditions is met, commands are read from the user's console.

>mapfile

Causes the listing output generated by **oml** to be written to the named file. If omitted, listing output is directed to the console.

options

Librarian options are specified as a hyphen followed by a string of characters which may not include white space. Current options include the following:

-oprefix Specifies that the output filenames for the command are to be formed by prepending the module name with **prefix.** Note that if a directory name is to be specified as a prefix, a trailing node separator (a slash under AmigaDOS) must be supplied on the prefix.

-s Causes a listing of the public symbols defined in the module to be included in the listing produced by the **l** command.

libfile

Specifies the name of the library file to be created or manipulated; this is the only command line field which must be present.

commands

These specify the actions to be performed by **oml** with respect to the specified library file. Commands are specified by a single character; if alphabetic, either upper or lower case may be used. They are separated from each other and from elements of file or module name lists by any white space. Current commands include:

r file-list

Replaces the named object files in the library, or adds them to the library if not already present. Note that replacement of existing modules in a library will work correctly only if the file name is the same as the module name. If the module does not exist in the library, this is unnecessary.

d module-list

Deletes the named modules from the library. Since modules without program unit names are assigned module names by the librarian, it may be necessary to obtain a listing (via the l command) in order to determine the assigned **$nnn** name for the module which is to be deleted.

x module-list

Extracts the named modules from the library, creating files of the same names. Note that if the module name includes a path name, the librarian will attempt to create a file with that name. All files are created in the current directory, unless the "-o" option is used; in that case, each module name is prefixed with the text specified on the "-o" option. The special character "*" may be

used to indicate that all of the modules in the library are to be extracted. **oml** will terminate execution if an attempt to extract a module is unsuccessful. Note that it is an error to specify the same name in both a replacement and an extraction list.

l

Causes generation of a listing of the modules in the library after all other requested operations have been performed. If the **-s** option is used on the command line which invokes **oml**, the listing will include the public symbols defined in each module as well as a list of the module names themselves.

@[filename]

Causes the remainder of command input to be read from **stdin**, or from the named file. Note: brackets are used to show that the file name is optional, and are not included if a file name is specified).

If replacement modules or deletions are specified, a new version of the library file will be built, provided that no errors are detected. This new version is created first as a temporary file; when it has been completely built, the original library file (if it existed) is deleted, and the temporary file renamed. This sequence insures that the original library file will not be affected if an error is detected. Modules are always included in a library in topologically sorted order, so that no backward references occur (except in the case of modules which reference each other, which are retained in the same order in which

they are encountered).

Warning messages are produced if (1) a module named in a deletion or extraction list was not found in the library or (2) if a second definition for a public symbol is encountered in one of the modules to be included.

EXAMPLES

The following examples illustrate the use of **oml.** Remember that replacing modules within a library file only works correctly if the module name is the same as the file name. Since a single object file is essentially a library of one module, **oml** can be used to find out what module name is included in the object file by using the following command:

```
oml name.o l
```

Building a New Library

Create a list of the file names of the object modules which will make up the library. The Lattice **extract** utility can be used to accomplish this, or any text editor. Then create the library using the following command:

```
oml new.lib r @name.lst
```

where **new.lib** is the name of the library to be created, and **name.lst** contains a list of the files to be included in the library. Note that creation of a new library is one occasion where the correspondence between file and module names is not required.

Extracting Modules from a Library

Use the following command to break out all of the modules from a library:

```
oml -o:object/ cfuncs.lib x *
```

Note that this command will be successful only if no module names in the library CFUNCS.L contain path names. A file for each of the modules in the library will be created in the directory **:object/** in this example.

Deleting Modules from a Library

Use the following command to delete modules from a library:

```
oml cfuncs.lib d tribe.o
```

This example deletes the module **tribe.o** from the library **cfuncs.lib.**

Listing the Modules in a Library

Use the following command to obtain a listing of the modules and symbols in a library file:

```
oml -s test.lib l
```

The listing may be saved to a file using I/O redirection:

```
oml >test.lst -s test.lib l
```

SEE ALSO

blink, lc

SECTION D
DATA NAMES

This section describes the public data names that are defined in the Lattice libraries.

C O N T E N T S

Disk Font Library Vector **Class: AMIGA**

NAME

DiskfontBase........Disk Font Library Vector

SYNOPSIS

```
extern long DiskfontBase;
DiskfontBase = OpenLibrary("diskfont.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines that interface with the ROM Kernel text functions that deal with new fonts. It must be initialized by an **OpenLibrary** call before any of the disk font functions documented in the Amiga ROM Kernel manual may be called. It is expected to contain the base address of the disk font library vector table.

DOSBase

AmigaDOS Library Vector **Class: AMIGA**

NAME

DOSBase.............AmigaDOS Library Vector

SYNOPSIS

```
extern long DOSBase;
DOSBase = OpenLibrary("dos.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines that interface with AmigaDOS system functions. It is initialized by an **OpenLibrary** call in the startup code and is expected to contain the base address of the AmigaDOS system library vector table. If you do not link with the startup module **c.o** and you make calls to AmigaDOS system functions or use Lattice features that call AmigaDOS system functions, then you must first initialize this location by calling **OpenLibrary.**

UNIX error number **Class: UNIX**

NAME

errno...............UNIX error number
sys_nerr............Number of UNIX error codes
sys_errlist.........UNIX error messages

SYNOPSIS

```
#include <error.h>

extern int errno;

extern int sys_nerr;

extern char *sys_errlist[];
```

DESCRIPTION

The external integer named **errno** is initialized to 0 at start-up time. Then if an error is detected by one of the standard library functions, a non-zero value is placed there. The standard library never resets **errno.**

Programmers typically use this information in two ways. In some cases, it is appropriate to check **errno** after a sequence of operations and abort if any error occurred along the way. In other cases, **errno** is checked periodically, and if it is non-zero, the appropriate corrective action is taken. Then the application program resets **errno** before beginning the next processing phase.

The **sys_nerr** and **sys_errlist** items are defined in a C source file named **syserr.c** and are used by the **perror** function to print messages that correspond to the code found in **errno.**

Note that even though error information is normally placed into **errno** by the standard library functions, application programs can also use this technique to indicate problems. However, you should be careful about adding new codes and messages just above the highest UNIX code currently defined, since new UNIX codes are occasionally added. Also, we recommend that you add application-dependent codes by extending the header file **error.h,** which contains symbolic definitions of the code numbers. The currently-defined codes are listed below:

UNIX error number **Class: UNIX**

SYMBOL	CODE	MEANING
EOSERR	-1	Operating system error
EPERM	01	User is not owner
ENOENT	02	No such file or directory
ESRCH	03	No such process
EINTR	04	Interrupted system call
EIO	05	I/O error
ENXIO	06	No such device or address
E2BIG	07	Arg list is too long
ENOEXEC	08	Exec format error
EBADF	09	Bad file number
ECHILD	10	No child process
EAGAIN	11	No more processes allowed
ENOMEM	12	No memory available
EACCES	13	Access denied
EFAULT	14	Bad address
ENOTBLK	15	Bulk device required
EBUSY	16	Resource is busy
EEXIST	17	File already exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Is not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	No more files (system)
EMFILE	24	No more files (process)
ENOTTY	25	Not a terminal
ETXTBSY	26	Text file is busy
EFBIG	27	File is too large
ENOSPC	28	No space left
ESPIPE	29	Seek issued to pipe
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math function argument error
ERANGE	34	Math function result is out of range

errno

UNIX error number **Class: UNIX**

SEE ALSO

perror

Graphics Library Vector **Class: AMIGA**

NAME

GfxBase.............Graphics Library Vector

SYNOPSIS

```
extern long GfxBase;
GfxBase = OpenLibrary("graphics.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines that interface with the ROM Kernel graphics system functions. It must be initialized by an **OpenLibrary** call before any of the graphics functions documented in the Amiga ROM Kernel manual may be called. It is expected to contain the base address of the graphics library vector table.

IntuitionBase

Intuition Library Vector **Class: AMIGA**

NAME

IntuitionBase.......Intuition Library Vector

SYNOPSIS

```
extern long IntuitionBase;
IntuitionBase = OpenLibrary("intuition.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines that interface with the Intuition system functions. It must be initialized by an **Open-Library** call before any of the functions documented in the Amiga Intuition manual may be called. It is expected to contain the base address of the Intuition library vector table.

FFP Library Vector **Class: AMIGA**

NAME

MathBase............FFP Library Vector

SYNOPSIS

```
extern long MathBase;
MathBase = OpenLibrary("mathffp.library",ver);
```

DESCRIPTION

This external location is used to interface with the Motorola Fast Floating Point library routines provided by Amiga. It is initialized by an **OpenLibrary** call when a program compiled with the FFP option performs the first floating point operation. It contains the base address of the FFP math library vector table. If you make direct calls to the Amiga FFP functions you must first initialize this location by calling **OpenLibrary**.

MathTransBase

FFP Trig Library Vector **Class: AMIGA**

NAME

MathTransBase.......FFP Trig Library Vector

SYNOPSIS

```
extern long MathTransBase;
MathTransBase = OpenLibrary("mathtrans.library",ver);
```

DESCRIPTION

This external location is used to interface with the Motorola Fast Floating Point format transcendental function library routines provided by Amiga. It is initialized by an **OpenLibrary** call when a program compiled with the FFP option performs the first floating point transcendental function call. It contains the base address of the FFP transcendental math library vector table. If you make direct calls to the Amiga FFP transcendental functions you must first initialize this location by calling **OpenLibrary.**

MSDOS File Pattern Flag **Class: LATTICE**

NAME

msflag..............MSDOS File Pattern Flag

SYNOPSIS

```
extern int msflag;
```

DESCRIPTION

This external integer is used by the file name pattern matching functions to specify AmigaDOS or MSDOS wildcard characters. If **msflag** is non-zero then MSDOS file name patterns are used. The default is to use AmigaDOS file name patterns.

SEE ALSO

dfind, getfnl

timedata

Time zone variables **Class: UNIX**

NAME

daylight............Daylight savings time flag
timezone............Timezone bias from GMT
tzname..............Timezone names
tzstn...............Standard time name
tzdtn...............Daylight time name

SYNOPSIS

```
extern int daylight;
extern long timezone;
extern char *tzname[2];
extern char tzstn[4];
extern char tzdtn[4];
```

DESCRIPTION

These variables are initialized by the **tzset** function and are used by the **localtime** function to adjust from Greenwich Mean Time to the local time.

The **daylight** item is non-zero if daylight saving time is currently in effect. The **timezone** value is the number of seconds that must be subtracted from GMT. The two **tzname** pointers point to **tzstn** and **tzdtn,** respectively. These strings contain the three-character names for standard time **(tzstn)** and daylight time **(tzdtn).**

SEE ALSO

localtime, tzset

Level 2 I/O buffer size **Class: LATTICE**

NAME

_bufsize...........Level 2 I/O buffer size

SYNOPSIS

```
extern int _bufsiz;
```

DESCRIPTION

This external integer is used by the level 2 I/O system to determine the size of the buffers allocated for level 2 files. This location is also used to determine the size of a buffer attached to a file with the **setbuf** function. In this case, **_bufsiz** must be set to the size of the buffer before **setbuf** is called.

Note that the buffer is not allocated when the file is opened. Instead, the first I/O operation causes the buffer to be allocated from the local memory pool if one has not been previously specified with **setbuf**. This means that if **_bufsiz** is changed between the open call and the first I/O operation, the size of the buffer allocated for the file will be the value of **_bufsiz** at the time of the I/O operation, not the value when the file was opened.

SEE ALSO

fopen, setbuf, _BUFSIZ

_fmode

Default level 2 I/O mode **Class: LATTICE**

NAME

_fmode..............Default level 2 I/O mode

SYNOPSIS

```
extern int _fmode;
```

DESCRIPTION

This external integer is used by the **fopen** function to determine the translation mode to use when the programmer does not specify a mode in the **fopen** call. For AmigaDOS it is set to 0x8000, which specifies untranslated mode.

SEE ALSO

fopen

Floating Point Error Code **Class: LATTICE**

NAME

_FPERR..............Floating Point Error Code

SYNOPSIS

```
extern int _FPERR;
```

DESCRIPTION

This location will contain a non-zero value after any low-level floating point operation encounters an error. Low-level operations include addition, subtraction, multiplication, division, comparison, and conversion from one number representation to another (e.g. float to double).

The error codes and their corresponding symbols from **math.h** and **math.mac** are:

SYMBOL	VALUE	MEANING
FPEUND	1	Underflow
FPEOVF	2	Overflow
FPEDVZ	3	Divide by zero
FPENAN	4	Not a valid number
FPECOM	5	Not comparable

When the error occurs, the low-level operation passes the appropriate error code to **CXFERR**, which must store the code in **_FPERR**. Note that **_FPERR** is never reset by any low-level operation.

SEE ALSO

CXFERR, _FPA

EXAMPLES

```
/*
*
* This example performs uses the division operation
* to stimulate floating point errors.
*
**/
#include <math.h>
#include <stdio.h>
extern int _FPERR;
main()
{
double a,b,c;

while(feof(stdin) == 0)
  {
  printf("Enter divisor:  ");
  if(scanf("%lf",&a) != 1) exit(0);
  printf("Enter dividend: ");
  if(scanf("%lf",&b) != 1) exit(0);
  _FPERR = 0;
  c = b / a;
  printf("_FPERR = %d\n",_FPERR);
  printf("%e / %e = %e\n\n",b,a,c);
  }
}
```

NAME

_MNEED..............Minimum Dynamic Memory Needed

SYNOPSIS

```
extern long _MNEED;   to reference default value
long _MNEED = n;      to set initial value
```

DESCRIPTION

_MNEED indicates the minimum number of bytes that must be provided as the so-called "heap space" for use by the various memory allocators. This space, whose size is rounded up to be an even multiple of **_mstep,** is the initial block of memory in the local memory pool.

If one of the memory allocators later determines that it needs more heap space, it will call upon **sbrk** to request additional memory from AmigaDOS. This memory then becomes a non-contiguous addition to the local pool.

Each time **rbrk** is called, all the memory in the local pool is returned to the system, and a request for the amount of memory in **_MNEED** is made of AmigaDOS.

Because of this technique, you can use **_MNEED** to guarantee at the outset that your program will have enough heap space. However, if you don't do anything with **_MNEED,** the memory allocators will still try to honor your space requests until AmigaDOS indicates that no more memory is available.

There are two ways to initialize this item. If your space needs are known at compilation time, you can simply declare **_MNEED** as an initialized public symbol. That declaration will then override the default

provided by the library, and the space will be obtained automatically by the startup routine (c.o). However, if you cannot determine your space needs until run time, you can load the appropriate value into **_MNEED** and then call **rbrk.** This, of course, should be only be done after all dynamic memory has been released via the appropriate de-allocator function. See the **rbrk** description for further information and cautions.

NAME

_mstep..............Memory Pool Increment Size

SYNOPSIS

```
extern int _mstep;
```

DESCRIPTION

This external integer is used by the memory allocation functions. It specifies the minimum amount of memory that will be allocated from the system for the local memory pool.

While additional memory is added to the local pool, it will not be contiguous with the memory already in the pool. If the additional amount is small, it can lead to severe fragmentation of the local pool. The memory allocation functions attempt to avoid this by rounding the amount needed up to the next multiple of the figure in **_mstep**.

This technique works well for small allocation requests. However, if your application requires mostly large blocks of memory, **_mstep** should be set to a small non-zero figure to allow for a more efficient allocation.

SEE ALSO

getmem, rlsmem

_OSERR

DOS Error Information   Class: AMIGA

NAME

_OSERR...............DOS Error Information
os_nerr..............Number of AmigaDOS error codes
os_errlist...........AmigaDOS error messages

SYNOPSIS

```
#include <dos.h>

extern int  _OSERR;

extern int os_nerr;

extern struct DOS_ERRS os_errlist[];
```

DESCRIPTION

The external integer named **_OSERR** contains error information returned by AmigaDOS after a system call has failed. In general, the Lattice library resets **_OSERR** at the beginning of any function that makes AmigaDOS system calls. Then if a system call fails during that function, the system error code is saved in **_OSERR.**

The AmigaDOS error number is mapped into an equivalent UNIX error number, which is placed in **errno.** If there is no appropriate UNIX number, **errno** will contain -1, defined symbolically as **EOSERR.** The function returns with a suitable error indication, which is usually -1 for functions that return integer values or NULL for functions that return pointers.

The **os_nerr** and **os_errlist** items are defined in a C source file named **oserr.c** and are used by the **poserr** function to print messages that correspond to the code found in **_OSERR.**

DOS Error Information **Class: AMIGA**

The following list applies to AmigaDOS 1.1 and is what we provide in **oserr.c**:

CODE	MEANING
103	Insufficient free store
104	Task table full
120	argument line invalid or too long
121	File is not an object module
122	Invalid resident library during load
202	Object in use
203	Object already exists
204	Directory not found
205	Object not found
206	Invalid window
210	Invalid stream component name
212	Object not of required type
213	Disk not validated
214	Disk write protected
215	Rename across devices attempted
216	Directory not empty
218	Device not mounted
220	Comment too big
221	Disk full
222	File is protected from deletion
223	File is protected from writing
224	File is protected from reading
225	Not a DOS disk
226	No disk in drive
209	Packet request type unknown
211	Invalid object lock
219	Seek error
232	No more entries in director

_OSERR

SEE ALSO

AmigaDOS Technical Reference, poserr

Directory separator character **Class: LATTICE**

NAME

_SLASH..............Directory separator character

SYNOPSIS

```
extern char _SLASH;
```

DESCRIPTION

This external character is used by various functions that construct file names. It specifies the character to be used for separating components of the directory path. For AmigaDOS, the default is a slash (/), while for MSDOS it is a backslash (\).

SEE ALSO

strmfn, strmfp

SECTION E
ENVIRONMENT VARIABLES

This section describes the AmigaDOS environment variables that are used by the Lattice C Compiler and related programs.

C O N T E N T S

NAME

include:............#include search path

SYNOPSIS

```
assign INCLUDE: include path
```

DESCRIPTION

This logical name is used by the compiler to find files that are included via the **#include** statement. The **include path** should be the disk and directory specification that describe the location of the header files. For example,

```
assign INCLUDE: "Lattice C:include"
```

indicates that the compiler should look for included files in the directory **include** on the disk whose volume name is **Lattice C.** Note that if any portion of the path contains spaces, then the entire path must be enclosed in double quotes.

The compiler accepts two forms of the **#include** statement, as follows:

```
#include <file>

#include "file"
```

When the first form is used, the compiler looks for **file** only in the directory associated with the **INCLUDE:** logical name. With the second form, the compiler looks in the current directory, then in any directories specified via the **-i** option, and finally in the directory associated with the **INCLUDE:** logical

name.

SEE ALSO

assign, lc, lc1

Compiler executable file path **Class: AMIGA**

NAME

lc:..................Compiler executable file path

SYNOPSIS

```
assign LC: program path
```

DESCRIPTION

This logical name is used by the driver program **lc** to locate the various programs required to compile and link source modules. This can be very useful on floppy disk based systems where there may not be enough online storage capacity to contain the compiler files in addition to other working files. By assigning **LC:** to the disk volume and directory containing the compiler and linker, **lc** will be able to locate and load the appropriate programs even if that disk volume isn't currently mounted. If required, you will be prompted to insert the necessary disk volume at the correct time.

For example,

```
assign LC: "Lattice C:"
```

will cause **lc** to load the compiler executable programs from the root level of the disk volume named **Lattice C**.

If the assignment is not made, **lc** looks for the programs in the default directory **lc/** in the root level of the system drive.

SEE ALSO

assign, lc

Library file path **Class: AMIGA**

NAME

lib:................Library file path

SYNOPSIS

```
assign LIB: library path
```

DESCRIPTION

This logical name is used by **lc** to locate the Lattice C startup routine **c.o** and the Lattice and Amiga libraries for the linker. If the assignment has been made, the driver program, **lc**, will specify that the linker look in the directory associated with the logical name for the startup code and the standard libraries. The directory may be on any disk volume. If the volume is not mounted, you will be prompted to insert the correct volume at the appropriate time. For example,

```
assign LIB: "Lattice C:lib"
```

specifies directory **lib/** on disk volume **Lattice C** as the location of the libraries and startup code.

If the assignment is not made, the linker will be directed to look in the subdirectory **lib/** of the path used to locate the linker. In other words, if the logical name **LC:** is assigned, the libraries will be assumed to be in the directory **LC:lib**. If **LC:** is not assigned, the directory **SYS:lc/lib** will be used.

SEE ALSO

assign, lc, lc1

Intermediate file path **Class: AMIGA**

NAME

quad:...............Intermediate file path

SYNOPSIS

```
assign QUAD: quadfile path
```

DESCRIPTION

This logical name is used to direct the compiler to place the intermediate file in some location other than the source file directory. By default, the compiler creates the intermediate, or quad, file in the same directory as the source file. You can use the **-q** option of the **lc** driver program or the **-o** option of lc1 to cause this file to be created somewhere else.

The **QUAD:** logical name is used by the driver program to assign a different default location for the intermediate file. For example,

```
assign QUAD: ram:
```

will cause the compiler to use ram disk for the intermediate file when it is invoked via the **lc** command.

SEE ALSO

assign, lc, lc1

SECTION F
FUNCTION NAMES

This section describes the public function names that are defined in the Lattice libraries.

C O N T E N T S

C O N T E N T S

C O N T E N T S

C O N T E N T S

C O N T E N T S

C O N T E N T S

Abort the current process **Class: ANSI**

NAME

abort...............Abort the current process

SYNOPSIS

```
#include <stdlib.h>

abort();
```

DESCRIPTION

This function aborts the current process and returns a completion code of 3 to the parent process. Also the message "Abnormal program termination" is sent to **stderr**. Level 2 I/O buffers are not flushed.

RETURNS

The function does not return.

SEE ALSO

exit, _exit

Absolute value **Class: ANSI**

NAME

abs................Absolute value
fabs...............Float/double absolute value
iabs...............Integer absolute value
labs...............Long integer absolute value

SYNOPSIS

```
#include <math.h>
ax = abs(x);

ad = fabs(d);
ai = iabs(i);
al = labs(l);

double ad,d;
int ai,i;
long al,l;
```

DESCRIPTION

These functions compute the absolute value of the various numeric data types.

The most general approach is to use the **abs** macro, which is defined in the **math.h** header file. This macro accepts any data type as its argument and generates in-line code to perform the conversion. The definition is:

```
#define abs(x) ((x)<0?-(x):(x))
```

To minimize code size, you can use one of the function calls listed above instead of the **abs** macro. However, you must be careful to choose the function that corresponds to the data type being converted. Note that **fabs** works with either a float or a double as its

Absolute value **Class: ANSI**

argument because float arguments are automatically promoted to double. Similarly, **iabs** will work with ints or shorts.

access

Check file accessibility **Class: UNIX**

NAME

access..............Check file accessibility

SYNOPSIS

```
#include <stdio.h>

ret = access(name,mode);

int ret;          return code
char *name;       file name
int mode;         access mode
```

DESCRIPTION

This function checks if a file is accessible in the way specified by **mode,** which follows the UNIX format:

```
0 => Check if file exists
1 => Check if file is executable
2 => Check if file is writable
3 => Check if file is writable and executable
4 => Check if file is readable
5 => Check if file is readable and executable
6 => Check if file is readable and writable
7 => Check if file is readable, writable, and executable
```

RETURNS

A return value of 0 indicates that access is allowed. If access is denied or the file cannot be found, -1 is returned. Additional error information can then be found in **errno** and **_OSERR.**

Check file accessibility **Class: UNIX**

SEE ALSO

chmod, errno, _OSERR

Get options from argument list **Class: LATTICE**

NAME

argopt.............Get options from argument list

SYNOPSIS

```
#include <stdlib.h>

optd = argopt(argc,argv,opts,argn,optc);

char *optd;       option data pointer
int argc;         argument count
char *argv[];     argument vector
char *opts;       options expecting data
int *argn;        next argument number (changed)
char *optc;       option character (changed)
```

DESCRIPTION

This function examines an argument list to find the next option argument, using the conventions similar to those of the UNIX "shell" command processor. These conventions are:

1. An option is an argument that begins with a slash (/) or a dash (i.e. a minus sign) and appears between the command verb (i.e. argv[0]) and the first non-option argument. The reason we recognize either a slash or a dash is that the former is a convention used by some AmigaDOS commands, while the latter has been used by UNIX for a long time.

2. The character immediately following the dash is called the "option character", and it may be followed by a character string known as the "option data".

3. If the option character appears in the **opts** string, then the data can be separated from the character by white space. In effect, this means that the data might be in the next **argv** entry if it does not follow the option character in the current entry.

4. A dash or slash followed by a blank or a dash indicates the end of the options.

Each time argopt is called, it will find the next option in the argument array and update the integer referenced by **argn**. On the first call, you should set this integer to 1, since **argv**[0] points to the command verb. The **argc** and **argv** items are normally the same as those passed to your main program, and they are not changed as a result of the **argopt** calls. The option character is returned in the byte referenced by **optc**, and the function returns a pointer to the option data string or to a null byte. If the next entry in **argv** is not an option, then the function returns a NULL pointer.

The **opts** item provides some flexibility in the way the option data is handled. If **opts** points to an empty string, then any option data must immediately follow the option character. However, if **opts** is not empty, then it lists the option characters that always have data. For those characters, the data can be preceded by white space in the command line. What this actually means is that **argopt** will look at the next entry in **argv** if the option character is not followed by a data string. If that next entry does not begin with a dash, then it is taken as the option data. See the examples below for clarification.

Get options from argument list **Class: LATTICE**

RETURNS

If the next argument is not an option, the function returns a NULL pointer. Otherwise, it returns a pointer to the option data, which will be an empty string if there was no data. Also, if an option was found, the character is placed into the byte referenced by **optc**, and **argn** is adjusted to index the next entry in **argv**.

SEE ALSO

main

EXAMPLES

```
/*
*
* Assume that this program is invoked by the following command line:
*
*       myprog -x -ypdq -z -g moo -g - blah
*
* The output will then be:
*
*       Option: x Data:
*       Option: y Data: pdq
*       Option: z Data:
*       Option: g Data: moo
*       Option: g Data:
*       Arg[8]: blah
*
*/
#include <stdlib.h>
char opts[] = "gx";
main(argc,argv)
int argc;
char *argv[];
{
```

Get options from argument list **Class: LATTICE**

```
char option,*odata;
int next;

for(next = 1; (odata = argopt(argc,argv,opts,&next,&option)) != NULL); )
  printf("Option: %c, Data: %s\n",option,odata);

for(; next < argc; next++)
        printf("Arg[%d]: %s\n",next,argv[next]);
}
```

asctime

Generate ASCII time string **Class: ANSI**

NAME

asctime............Generate ASCII time string

SYNOPSIS

```
#include <time.h>

s = asctime(t);

char *s;         points to time string
struct tm *t;    points to time structure
```

DESCRIPTION

This function converts a time structure into an ASCII string of exactly 26 characters having the form:

```
DDD MMM dd hh:mm:ss YYYY\n\0
```

where **DDD** is the day of the week, **MMM** is the month, **dd** is the day of the month, **hh:mm:ss** is the hour:minute:seconds, and **YYYY** is the year. An example is:

```
Wed Sep Ø4 15:13:22 1985\n\Ø
```

The time pointer returned by the function refers to a static data area that is shared by both **ctime** or **asctime.** The time structure argument **t** is usually returned by the **gmtime** or **localtime** function.

SEE ALSO

gmtime, localtime

Generate ASCII time string **Class: ANSI**

EXAMPLES

```
#include <time.h>
#include <stdio.h>

main()
{
struct tm *tp;
long t;

time(&t);
tp = localtime(&t);
printf("Current time is %s\n",asctime(tp));
}
```

Assert program validity **Class: ANSI**

NAME

assert..............Assert program validity
_assert............Failure exit for assert

SYNOPSIS

```
#include <assert.h>

assert(x);
_assert(exp,file,line);

char *exp;      failing expression
char *file;     source file name
char *line;     source line number
```

DESCRIPTION

The **assert** macro tests an expression **x** for validity (non-zero value). If the expression is Ø, then the macro calls the **_assert** function with the expression in text form plus the source file name and line number, also as text strings. The default version of **_assert** prints a message on **stderr** and aborts with an exit code of 1. You can replace this function with one of your own if you require some other action when an assertion fails. The source code is supplied in the compiler package.

Note that the **assert.h** header file must be included in your program in order to define the macro. Also, the file contains two versions of the macro. If the symbol **NDEBUG** is defined, then a null version of the macro is used; otherwise the normal code-generating version applies. This allows you to strip the assertion code from your program without removing the **assert** calls. To do this, simply define **NDEBUG** in one of your header files or on the compiler command line via the **-d**

option. In the former case, the header file containing the **NDEBUG** definition must be included before **assert.h.**

EXAMPLES

```
/* Make sure integer x is positive */

postest(x)
int x;
{
assert(x >= 0);
}
```

atof

Convert ASCII to float **Class: ANSI**

NAME

atof...............Convert ASCII to float

SYNOPSIS

```
#include <math.h>

d = atof(p);

double d;        floating point result
char *p;         input string pointer
```

DESCRIPTION

This function converts an ASCII input string into a double value. The string can contain leading white space and a plus or minus sign, followed by a valid floating point number in normal or scientific notation. If scientific notation is used, there can be no white space between the number and the exponent. For example,

```
123.456e-53
```

is a valid number in scientific notation.

EXAMPLES

```
/*
*
* This program tests the atof function.
*
*/
#include <stdio.h>
#include <math.h>

main()
```

Convert ASCII to float **Class: ANSI**

```
{
char buff[80];
double d;

while(1)
        {
        printf("\nEnter a number: ");
        if(gets(buff) == NULL) exit(0);
        if(buff[0] == '\0') exit(0);
        d = atof(buff);
        printf("%e\n",d);
        }
}
```

Convert ASCII to integer **Class: UNIX**

NAME

atoi................Convert ASCII to integer
atol................Convert ASCII to long integer

SYNOPSIS

```
#include <stdlib.h>

x = atoi(s);
y = atol(s);

int x;
long y;
char *s;
```

DESCRIPTION

These functions convert ASCII strings into normal or long integers. The string must have the form:

[whitespace][sign]digits

where **[whitespace]** indicates optional leading white space, **[sign]** indicates an optional + or - sign character, and **digits** is a contiguous string of digit characters. Once the digit portion is reached, the conversion continues until a non-digit character is hit. No check is made for integer overflow.

RETURNS

As noted above.

SEE ALSO

atof, stcd_i, stcd_l

Level 3 memory allocation **Class: ANSI**

NAME

calloc..............Allocate and clear Level 3 memory
free................Free Level 3 memory
malloc..............Allocate Level 3 memory
realloc.............Re-allocate Level 3 memory

SYNOPSIS

```
#include <stdlib.h>

b = calloc(nelt,esize);
b = malloc(n);
nb = realloc(b,n);
error = free(b);

char *b;                   block pointer
unsigned nelt;             number of elements
unsigned esize;            element size
unsigned n;                number of bytes
char *nb;                  new block pointer
int error;                 0 for success, -1 for failure
```

DESCRIPTION

These functions form Level 3 of Lattice's layered memory allocation system. This level is fully compatible with UNIX and with the ANSI standard.

The **malloc** function allocates a block that is **n** bytes long and is aligned in such a way that you can cast the block pointer to any pointer type. If the block cannot be allocated, a NULL pointer is returned. The **calloc** function uses **malloc** to get a block whose size in bytes is given by

```
n = nelts * esize;
```

Then the block is cleared to zeroes. Like **malloc**, **calloc** returns a NULL pointer if the block cannot be allocated.

A call to **realloc** will obtain a new block whose size is **n** bytes. Then it copies the old block **b** to the new block **nb** and releases the old block. If **n** is greater than the old block size, the excess space is cleared to zeroes. If it is less than the old block size, only the first **n** bytes are copied. As described below, you can, under certain circumstances, re-allocate a block that has been freed.

The **free** function releases a block that was previously obtained via **calloc**, **malloc**, or **realloc**. For compatibility with some versions of UNIX, the block is not actually returned to the free space pool until the next time you call **calloc**, **malloc**, **realloc**, or **free**. Then if that next call is to **realloc** and the block being re-allocated is the one that was just freed, **realloc** will proceed correctly. In other words, you can ask **realloc** to re-allocate a block that was freed as long as you have not called **calloc**, **malloc**, or **realloc** in the meantime. Believe it or not, many UNIX programmers have come to depend on this rather bizarre behavior.

RETURNS

For **calloc**, **malloc**, and **realloc**, a NULL pointer is returned if there is not enough space for the requested block. For **free**, 0 is returned if the block was successfully released; otherwise, the function returns -1.

CAUTIONS

These functions are called "Level 3 memory allocation" because they call upon Level 2 functions **getmem** and **rlsmem.** You should be aware that the Level 1 function **rbrk** frees all Level 2 and Level 3 blocks. Be careful!

SEE ALSO

getmem, rlsmem, sbrk, rbrk

EXAMPLES

```
/*
*
* This program builds linked lists of text strings obtained from the
* standard input file.
*
*/
#include <stdio.h>
#include <stdlib.h>
/*
*
* These elements are linked together to form the text string list.
*
*/
struct LIST
  {
  struct LIST *next;     /* forward linkage */
  char text[2];          /* minimum text string */
  };
/*
*
* Main program
*
*/
main()
{
```

```
struct LIST *p,*q,*list = NULL;
char b[256];
int x;
/*
* Build the list
*/
while(1)
  {
  printf("\nBegin new group...\n");
  for(q = (struct LIST *)(&list);;q = p)
    {
    printf("Enter a text string: ");
    if(gets(b) == NULL) break;
    if(b[Ø] == '\Ø') break;
    x = sizeof(struct LIST) - 2 + strlen(b) + 1;
    p = (struct LIST *)malloc(x);
    if(p == NULL)
      {
      printf("No more memory\n");
      break;
      }
    p->next = q->next;
    strcpy(p->text,b);
    }
/*
* Print the list
*/
  printf("\n\nTEXT LIST...\n");
  for(p = list; p != NULL; p = p->next)
    {
    printf("%s",p->text);
    free(p);
    }
  }
}
```

Get floating point limits **Class: ANSI**

NAME

ceil................Get ceiling of a real number
floor...............Get floor of a real number

SYNOPSIS

```
#include <math.h>

x = ceil(y);
x = floor(y);

double x,y;
```

DESCRIPTION

These functions return the integral values that are nearest to the specified real number. For **ceil,** the return is the next higher integer, while **floor** returns the next lower integer.

CAUTIONS

Note that even though these functions return integral values, the results are still real numbers.

EXAMPLES

```
#include <math.h>

double r;

r = ceil(523.96);        /* r contains 524.0 */
r = floor(523.96);       /* r contains 523.0 */
```

chdir

Change current directory **Class: UNIX**

NAME

chdir...............Change current directory

SYNOPSIS

```
#include <stdio.h>

error = chdir(path);
int error;        0 if successful
char *path;       points to new directory path string
```

DESCRIPTION

This function changes the current directory to the specified path. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the return value is non-zero, then the operation failed. An AmigaDOS error code will be in **_OSERR,** and a UNIX error code will be in **errno.**

SEE ALSO

mkdir, rmdir, getcd, getcwd, errno, _OSERR

Check for Break character **Class: AMIGA**

NAME

chkabort...........Check for Break character

SYNOPSIS

```
void chkabort();
```

DESCRIPTION

This function forces immediate checking for the break characters CTRL-C and CTRL-D. Normally this check only occurs when level 1 I/O is performed. The **chkabort** function provides a mechanism for detecting the break characters at other times. This can be important in programs that do little or no I/O processing.

If either break key is detected, the break trap is executed. The break trap is simply a function that gets called whenever the user keys CTRL-C or CTRL-D. If the break trap returns a zero value, control returns to the calling function. Otherwise the program terminates. The default handler displays a requester allowing the user to abort the program. It may be replaced via calls to **onbreak** or **signal**.

RETURNS

There is no return value.

SEE ALSO

signal, onbreak

chkml

Check for largest memory block **Class: LATTICE**

NAME

chkml...............Check for largest memory block

SYNOPSIS

```
#include <stdlib.h>

size = chkml();
long size;
```

DESCRIPTION

This function returns the size, in bytes, of the largest block that is currently available in the level 2 memory pool without calling upon the operating system to supply additional heap space.

CAUTIONS

The return value is a long integer, so the function must be declared appropriately.

SEE ALSO

getmem, getml, rlsmem, rlsml, sizmem

Check Level 1 file handle **Class: LATTICE**

NAME

chkufb..............Check Level 1 file handle

SYNOPSIS

```
#include <ios1.h>

ufb = chkufb(fh);

struct UFB *ufb;          pointer to UNIX file block
int fh;                   file handle
```

DESCRIPTION

This function checks if an AmigaDOS file handle is currently associated with a Level 1 file. Normally it is used internally by **open, close, read, write, lseek,** and **tell.** The UFB structure is defined in header file **ios1.h.** For AmigaDOS this structure is simply two integers. The first contains the mode flags specified in the call to the **open** function. The second integer contains the file handle. The external name **_ufbs** refers to an array of UFB structures, and the external integer **_nufbs** indicates how many structures are in the array. Normally this value is twenty.

RETURNS

If no UFB is currently attached to the file handle, a null pointer is returned.

chmod

Change file protection mode **Class: UNIX**

NAME

chmod...............Change file protection mode

SYNOPSIS

```
#include <stdio.h>

error = chmod(name,mode);

int error;        error code
char *name;       file name
int mode;         protection mode
```

DESCRIPTION

This function changes a file's protection mode. It is compatible with UNIX, although AmigaDOS also provides separate delete protection for each file. The **mode** argument should be formed by ORing any combination of the following symbols which are defined in **fcntl.h:**

VALUE	MEANING
S_IWRITE	Write permission
S_IREAD	Read permission
S_IEXECUTE	Execute permission
S_IDELETE	Delete permission

RETURNS

If the operation is successful, the function returns Ø. Otherwise it returns -1 and places error information in **errno** and **_OSERR.**

Change file protection mode **Class: UNIX**

SEE ALSO

access, errno, _OSERR

EXAMPLES

```
/*
*
* This example changes file "xyz/pdq.x" so it
* can be read and written.
*
*/
#include <stdlib.h>

if(chmod("xyz/pdq.x",S_IWRITE | S_IREAD))
  perror("Change mode");
```

clearerr,clrerr

Clear Level 2 I/O error flag **Class: ANSI**

NAME

```
clearerr...........Clear Level 2 I/O error flag
clrerr.............Clear Level 2 I/O error flag
```

SYNOPSIS

```
#include <stdio.h>

clrerr(fp);
clearerr(fp);

FILE *fp;           file pointer
```

DESCRIPTION

These functions clear the error flag associated with the specified Level 2 file. Once set, the error flag forces an EOF return any time the file is accessed until the flag is reset.

Note that **clearerr** is a macro, while **clrerr** is a function. The former is provided for compatibility with some older versions of UNIX.

RETURNS

None.

SEE ALSO

fopen

Close a Level 1 file **Class: UNIX**

NAME

close..............Close a Level 1 file

SYNOPSIS

```
#include <fcntl.h>

error = close(fh);

int error;        non-zero if error
int fh;           file handle
```

DESCRIPTION

This function closes a Level 1 file that was previously opened via the **open** function. Any pending output is completed and the file directory is updated.

All Level 1 files are automatically closed when your program terminates, but it is good programming practice to close a file when you are finished with it. One reason for doing this is to free up the operating system resources (e.g. control blocks and buffers) that are allocated for the file while it remains open.

RETURNS

The function returns Ø if it is successful. Otherwise, it returns -1 and places additional error information into **errno** and **_OSERR**.

SEE ALSO

errno, open, _OSERR

EXAMPLES

See the **open** function.

Create a Level 1 file **Class: UNIX**

NAME

creat...............Create a Level 1 file

SYNOPSIS

```
#include <fnctl.h>

fh = creat(name,prot);

int fh;          file handle
char *name;      file name
int prot;        protection mode
```

DESCRIPTION

This function is exactly the same as calling the **open** function in the following way:

```
open(name,O_WRONLY | O_TRUNC | O_CREAT , prot)
```

In other words, the file is created if it doesn't exist and truncated if it does exist. Then it is opened for writing. The protection mode can be any of the following:

VALUE	**MEANING**
S_IWRITE	Write permission
S_IREAD	Read permission
S_IWRITE ¦ S_IREAD	Write and read permission
Ø	Write and read permission

In the current AmigaDOS implementation the protection mode is ignored.

RETURNS

If the operation is successful, the function returns a file handle, which is an integer equal to or greater than Ø. Otherwise it returns -1 and places error information in **errno** and **_OSERR.**

SEE ALSO

errno, _OSERR, chgfa, chmod, close, open

Convert time value to string **Class: ANSI**

NAME

ctime...............Convert time value to string

SYNOPSIS

```
#include <time.h>

s = ctime(t);
char *s;          points to time string
long *t;          points to time value
```

DESCRIPTION

This function converts a GMT time value to an ASCII string of exactly 26 characters having the form:

```
DDD MMM dd hh:mm:ss YYYY\n\0
```

where **DDD** is the day of the week, **MMM** is the month, **dd** is the day of the month, **hh:mm:ss** is the hour:minute:seconds, and **YYYY** is the year. An example is:

```
Wed Sep 04 15:13:22 1985\n\0
```

The time pointer returned by the function refers to a static data area that is shared by both **ctime** or **asctime**.

The time value argument **t** must point to a long integer that is the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. Normally this value is obtained from the **time** function. Note that **ctime** converts this value back into local time by calling **tzset** and then subtracting the contents of **timezone**.

Convert time value to string **Class: ANSI**

CAUTIONS

Note that **t** is a pointer to a long integer. A common error is to pass the integer itself instead of the pointer. Observe the use of the ampersand (&) operator in the example above.

SEE ALSO

asctime, gmtime, localtime, time

EXAMPLES

```
#include <time.h>
#include <stdio.h>

main()
{
long t;

time(&t);
printf("Current time is %s\n",ctime(&t));
}
```

Low-level float error exit **Class: LATTICE**

NAME

CXFERR..............Low-level float error exit

SYNOPSIS

```
#include <math.h>

CXFERR(code);
int code;
```

DESCRIPTION

This function is called when an error is detected by one of the low-level floating point routines, such as the arithmetic operations. Higher-level routines, such as the trigonometric functions, use the more sophisticated **matherr** function.

Users can replace this error trap with an application-dependent routine, as long as they still store the error code into the global integer **_FPERR**. This is necessary because some of the math functions check **_FPERR** to see if low-level errors occurred.

The error code passed to **CXFERR** indicates the type of floating point anomaly that occurred, as follows:

SYMBOL	VALUE	MEANING
FPEUND	1	Underflow
FPEOVF	2	Overflow
FPEZDV	3	Divide by zero
FPENAN	4	Not a number
FPECOM	5	Not comparable

These codes are defined in **math.h**.

CXFERR

Low-level float error exit **Class: LATTICE**

SEE ALSO

_FPERR, _FPA, matherr

Close an AmigaDOS file **Class: AMIGA**

NAME

dclose.............Close an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

error = dclose(fh);

int error;        0 for success, -1 for error
int fh;           file handle
```

DESCRIPTION

This function closes an AmigaDOS file that was opened via **dcreat, dcreatx,** or **dopen.** AmigaDOS will not automatically close these files for you. One reason for doing this is to free up the operating system resources (e.g. control blocks and buffers) that are allocated for the file while it remains open.

If your program terminates normally, the run time support library will automatically close all files opened via level 1 or level 2 I/O calls.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR.**

SEE ALSO

errno, _OSERR, dcreat, dcreatx, dopen, dread, dwrite, dseek

dcreat,dcreatx

Create AmigaDOS file **Class: AMIGA**

NAME

dcreat..............Create or truncate AmigaDOS file
dcreatx.............Create new AmigaDOS file

SYNOPSIS

```
#include <dos.h>

fh = dcreat(name,fatt);
fh = dcreatx(name,fatt);

int fh;          file handle (-1 for error)
char *name;      file name
int fatt;        file attribute
```

DESCRIPTION

These functions create and open an AmigaDOS file, returning the file handle. The **dcreat** operation will truncate the file if it already exists, or create the file if it does not exist. On the other hand, **dcreatx** will fail if the file already exists.

The file attribute argument is accepted to provide source level compatibility with other systems, but is ignored under AmigaDOS.

RETURNS

If the operation is successful, the function returns a file handle. Otherwise it returns -1 and places error information in **errno** and **_OSERR.**

SEE ALSO

errno, _OSERR, dopen, dclose, dread, dwrite,

Find directory entry **Class: AMIGA**

NAME

dfind..............Find first directory entry
dnext..............Find next directory entry

SYNOPSIS

```
#include <dos.h>

error = dfind(info,name,attr);
error = dnext(info);

int error;                  0 if successful
struct FILEINFO *info;      file information area
char *name;                 file name or pattern
int attr;                   file attribute bits
```

DESCRIPTION

These functions search a directory for entries that match the specified file name or file name pattern. The **dfind** function locates the first matching file. Then successive calls to **dnext** locate additional matching files. Each **dnext** call must be given the file information that was returned on the preceding call to **dfind** or **dnext**.

The **name** argument must be a null-terminated string specifying the drive, path, and name of the desired file. The drive and path can be omitted, in which case the current directory will be searched. You can use the AmigaDOS wildcard characters as defined in the AmigaDOS Users Manual for pattern matching in the name portion. For example, "xy#?.b" will locate files in the current directory that begin with "xy" and have "b" as their extension. Setting the external integer location **msflag** to a nonzero value will cause all

dfind,dnext

Find directory entry **Class: AMIGA**

subsequent calls to **dfind** and **dnext** to use the MSDOS * and ? characters for pattern matching in the name portion.

The **attr** argument specifies which file types are to be included in the search. If **attr** is zero, the search will include only normal files. Otherwise the search will also include directories.

The **info** argument points to a file information structure as defined in the **dos.h** header file. For AmigaDOS, this is the same as the AmigaDOS FileInfoBlock structure:

```
struct FileInfoBlock {
    LONG    fib_DiskKey;
    LONG    fib_DirEntryType;
    char    fib_FileName[108];
    LONG    fib_Protection;
    LONG    fib_EntryType;
    LONG    fib_Size;
    LONG    fib_NumBlocks;
    struct DateStamp fib_Date;
    char    fib_Comment[116];
};
```

RETURNS

If the operation is successful, a value of 0 is returned. Otherwise, the return value is -1, and further error information can be found in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

Open an AmigaDOS file **Class: AMIGA**

NAME

dopen...............Open an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

fh = dopen(name,mode);

int fh;          file handle (-1 for error)
char *name;      file name
int mode;        access mode
```

DESCRIPTION

This function opens an AmigaDOS file and returns the file handle. The **mode** argument must be a mode supported directly by AmigaDOS and defined in the Amiga header file **libraries/dos.h.**

RETURNS

If the operation is successful, the function returns a file handle. Otherwise it returns -1 and places error information in **errno** and **_OSERR.**

SEE ALSO

errno, _OSERR, open, dcreat, dcreatx, dclose, dread, dwrite, dseek

drand

Generate random numbers **Class: UNIX**

NAME

```
drand48.............Random double (internal seed)
erand48.............Random double (external seed)

lrand48.............Random positive long (internal
seed)
nrand48.............Random positive long (external
seed)

mrand48.............Random long (internal seed)
jrand48.............Random long (external seed)

srand48.............Set high 32 bits of internal seed
seed48..............Set all 48 bits of internal seed
lcong48.............Set linear congruence parameters
```

SYNOPSIS

```
#include <math.h>

x = drand48();
x = erand48(seed);

y = lrand48();
y = nrand48(seed);

z = mrand48();
z = jrand48(seed);

srand48(hseed);
pseed = seed48(seed);
lcong48(parm);

double x;         random double
long y;           random positive long
long z;           random long
short seed[3];    seed value (high bits in seed[0])
```

Generate random numbers **Class: UNIX**

```
long hseed;      high 32 bits of seed value
short *pseed;    pointer to internal seed
short parm[7];   parameters
```

DESCRIPTION

These functions generate various types of random numbers using the linear congruential algorithm and 48-bit arithmetic. The normal functions **drand48, lrand48,** and **mrand48** use an internal 48-bit storage area for the seed value. Special versions **erand48, jrand48,** and **nrand48** are provided for cases where several seeds are in use at the same time, in which case the user specifies the seed on each function call.

The **drand48** and **erand48** functions return double values uniformly distributed over the interval from Ø.Ø up to but not including 1.Ø.

The **lrand48** and **nrand48** functions return non-negative long integers uniformly distributed over the interval from Ø to 2**31-1.

The **mrand48** and **jrand48** functions return signed long integers uniformly distributed over the interval from -2**31 to 2**31-1.

The **srand48** and **seed48** functions allow initialization of the internal 48-bit seed to something other than the default. For **srand48** the specified long value is copied into the high 32 bits of the seed, and the low 16 bits are set to Øx33ØE. For **seed48** the entire 48 bits are loaded from the specified array, and the function returns a pointer to the internal seed array.

The **lcong48** function allows a much more intricate

drand

Generate random numbers **Class: UNIX**

initialization of the linear congruential algorithm. The algorithm is of the form:

$$X[n+1] = (a * X[n] + c) \bmod m$$

where **m is 2**48 and the default values for a** and **c** are Ø x5DEECE66D and ØxB, respectively. The array passed to **lcong48** is structured as follows:

PARAMETER	VALUE
parm[Ø]	Bits 47-32 of value X[n]
parm[1]	Bits 31-16 of value X[n]
parm[2]	Bits 15-ØØ of value X[n]
parm[3]	Bits 47-32 of value a
parm[4]	Bits 31-16 of value a
parm[5]	Bits 15-ØØ of value a
parm[6]	value c

Whenever **seed48** is called, **a** and **c** are reset to their default values.

RETURNS

As noted above.

SEE ALSO

rand, srand

Read and write AmigaDOS files **Class: AMIGA**

NAME

dread..............Read from an AmigaDOS file
dwrite.............Write to an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

count = dread(fh,buffer,length);
count = dwrite(fh,buffer,length);

int count;        actual bytes read or written
int fh;           file handle
char *buffer;     data buffer
int length;       number of bytes to read or write
```

DESCRIPTION

These functions read or write an AmigaDOS file whose handle was returned by **dcreat**, **dcreatx**, or **dopen**. Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult **errno** and **_OSERR**. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check **errno** and **_OSERR** just in case some malfunction caused the short count.

RETURNS

If the operation is successful, the function returns the actual number of bytes transferred. Otherwise it

returns -1 and places error information in **errno** and **_OSERR.**

SEE ALSO

errno, _OSERR, dcreat, dcreatx, dopen, dclose, dseek

Re-position AmigaDOS file **Class: AMIGA**

NAME

dseek...............Re-position AmigaDOS file

SYNOPSIS

```
#include <dos.h>

apos = dseek(fh,rpos,mode);

long apos;        actual file position
int fh;           file handle
long rpos;        relative file position
int mode;         seek mode
```

DESCRIPTION

This function re-positions an AmigaDOS file whose handle was returned by **dcreat, dcreatx,** or **dopen.** The seek mode is the same as for **lseek** as follows:

Mode 0 Position is relative to beginning of file.

Mode 1 Position is relative to current file location.

Mode 2 Position is relative to end of file.

Note that for modes 1 and 2 **rpos** can be positive or negative, but **apos** is always the actual (positive) position relative to the beginning of file.

RETURNS

If the operation is successful, the function returns the actual file position, which is a long integer. Otherwise it returns -1L and places error information in **errno** and **_OSERR.**

SEE ALSO

errno, _OSERR, dcreat, dcreatx, dopen, dclose, dread, dwrite

Convert float to string **Class: UNIX**

NAME

```
ecvt................Convert float to string
fcvt................Convert float to string
```

SYNOPSIS

```
#include <math.h>

s = ecvt(v,dig,decx,sign);
s = fcvt(v,dec,decx,sign);

char *s;          string pointer
double v;         floating point value
int dig;          number of digits
int dec;          number of decimal places
int *decx;        pointer to decimal index (returned)
int *sign;        pointer to sign indicator
```

DESCRIPTION

These functions convert a floating point number into an ASCII character string consisting of digits only and terminated by a null character.

For **ecvt,** the second argument indicates the total number of digits that should be generated, while for **fcvt** it indicates how many digits should be generated to the right of the decimal place. If the floating point value contains fewer significant digits, zeroes are appended. If there are too many significant digits, the low order (right-most) digit is rounded.

The **decx** argument points to an integer that will receive a value indicating where the decimal point should be placed in the string. For example, an index value of 3 indicates that the decimal point should be placed just after the third character in the string. A

value of zero means that the decimal point is just before the first character. If the index is negative, it indicates the number of zeroes that are between the decimal point and the first character. For example, -3 means that there are three zeroes between the decimal point and the beginning of the string.

The **sign** argument points to an integer that will be non-zero if **v** is negative.

EXAMPLES

```
#include <stdlib.h>

int decx,sign;
char *string;

string = ecvt(3.1415926535,10,&decx,&sign);

/*
*   string  =>  "3141592654"
*   decx    =>  1
*   sign    =>  0
*/

string = fcvt(3.1415926535,10,&decx,&sign);

/*
*   string  =>  "31415926535"
*   decx    =>  1
*   sign    =>  0
*/
```

Terminate program execution **Class: ANSI**

NAME

exit................Terminate with clean-up
_exit...............Terminate with no clean-up

SYNOPSIS

```
#include <stdlib.h>

exit(code);
_exit(code);
```

DESCRIPTION

These functions terminate execution of the current program and return control to the parent program. Use **exit**, for a graceful termination, which means that all pending output buffers are written and all files are explicitly closed. The **_exit** function terminates immediately without writing output buffers or closing level 2 files. Generally, this latter form is used only in emergency situations when you don't care if some output data is lost.

In either case, files opened via **open**, **creat**, or **creatx** will be closed. However, files opened via **dopen**, **dcreat**, or **dcreatx** will not be closed.

The parameter **code** is a value from 0 to 255 that gets passed back to the parent. By convention, a value of zero indicates success.

RETURNS

This function does not return.

EXAMPLES

```
/*
*
* This example shows how you would abort a program
* if it is not called with a valid input file name.
*
*/
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
FILE *f;

if(argc > 1)
  {
  f = fopen(argv[1],"r");
  if(f == NULL)
    {
    fprintf(stderr,"Can't open file \"%s\"\n",argv[1]);
    exit(1);
    }
  }
else
  {
  fprintf(stderr,"No file specified\n");
  exit(1);
  }
```

Exponential functions **Class: ANSI**

NAME

```
exp.................Exponential function
log.................Natural logarithm function
log10...............Base 10 logarithm function
pow.................Power function
sqrt................Square root function
```

SYNOPSIS

```
#include <math.h>

r = exp(x);
r = log(x);
r = log10(x);
r = pow(x,y);
r = sqrt(x);

double r, x, y;
```

DESCRIPTION

The **exp** function raises the natural logarithm base E to the **x** power, and **pow** raises **x** to the **y** power. For **pow,** the **x** value must be an integer if it is negative. If it is not integral, **matherr** is called with a DOMAIN error.

The **log** and **log10** functions take the base E and base 10 logarithm, respectively. Each of these, as well as **sqrt,** requires a positive argument. If a negative argument is supplied, **matherr** will be called with a DOMAIN error.

SEE ALSO

fclose,fcloseall

Close a Level 2 file **Class: ANSI**

NAME

```
fclose.............Close a Level 2 file
fcloseall..........Close all Level 2 files
```

SYNOPSIS

```
#include <stdio.h>

ret = fclose(fp);
num = fcloseall();

int ret;          return code
int num;          number of files closed
FILE *fp;         file pointer for file to be closed
```

DESCRIPTION

The **fopen** function completes the processing of a Level 2 file and releases all related resources. If the file was being written, any data which has accumulated in the buffer is written to the file, and the level 1 **close** function is called for the associated file descriptor. The buffer associated with the file block is freed. Even though **fclose** is automatically called for all open files when your program terminates or calls **exit,** it is good programming practice to close your own files explicity. The last buffer is not written until **fclose** is called, and so data may be lost if an output file is not properly closed.

The **fcloseall** function closes all Level 2 files that were open and returns that number. However, if an error occurs on any file, **fcloseall** continues to close the other files and then returns a value of -1.

Close a Level 2 file **Class: ANSI**

RETURNS

```
ret = -1 if error
    = Ø if successful

num = -1 if error
    = number of files closed if no errors
```

If -1 is returned, additional error information can be found in **errno** and **_OSERR.**

CAUTIONS

Remember that **fcloseall** closes the standard files **stdin, stdout,** and **stderr,** This means, for example, that functions such as **printf** and **perror** will fail after you call **fcloseall.**

SEE ALSO

fopen, errno, _OSERR

fdopen

Attach L1 file to L2 **Class: UNIX**

NAME

fdopen..............Attach L1 file to L2

SYNOPSIS

```
#include <stdio.h>

fp = fdopen(fh,mode);

FILE *fp;           file pointer
int fh;             file handle
char *mode;         access mode
```

DESCRIPTION

This function attaches a level 1 file to level 2. In other words, if you have used **open** to prepare a file for level 1 I/O processing, you can subsequently use level 2 I/O with that file via **fdopen.** The file handle is the value returned by the **open** function, and the access mode has the same form as described for **fopen.**

RETURNS

If the operation is successful, the function returns a non-null file pointer. Otherwise it returns a null pointer and places error information in **errno** and **_OSERR.**

SEE ALSO

fopen, errno, _OSERR

Check Level 2 file status **Class: ANSI**

NAME

feof...............Check for Level 2 end-of-file
ferror.............Check for Level 2 error

SYNOPSIS

```
#include <stdio.h>

ret = feof(fp);
ret = ferror(fp);

int ret;          non-zero if condition is true
FILE *fp;         file pointer
```

DESCRIPTION

These functions generate a non-zero value if the indicated condition is true for the specified file.

RETURNS

The return value is 0 if the condition is false, that is, no error for **ferror** or no end-of-file for **feof.** If the condition is true, a non-zero value is returned.

CAUTIONS

These functions are implemented as macros. Also, they do not check that **fp** is a valid file pointer.

fflush,flushall

Flush Level 2 output buffer **Class: ANSI**

NAME

fflush.............Flush a Level 2 output buffer
flushall...........Flush all Level 2 output buffers

SYNOPSIS

```
#include <stdio.h>

ret = fflush(fp);
num = flushall();

FILE *fp;          file pointer
int ret;           return code
int num;           number of open files
```

DESCRIPTION

The **fflush** macro flushes the output buffer of the specified Level 2 file. That is, it writes the buffer if the file is opened for output and the buffer contains any pending data. If an error occurs, the return value is EOF and the appropriate error code is placed into **errno**.

The **flushall** function flushes all Level 2 output buffers and returns the number of Level 2 files that are open. If an error occurs, the function continues to flush the remaining files and then returns a value of -1.

RETURNS

```
ret = 0 if successful
    = EOF if error

num = number of open files if no error
    = -1 if error
```

Flush Level 2 output buffer **Class: ANSI**

In the event of a -1 return, error information can be found in **errno** and _OSERR.

SEE ALSO

fopen, fclose, errno, _OSERR

Get a character **Class: ANSI**

NAME

fgetc..............Get a character from a file
getc...............Get a character from a file
getchar............Get a character from stdin
fgetchar...........Get a character from stdin

SYNOPSIS

```
#include <stdio.h>

c = fgetc(fp);
c = getc(fp);
c = getchar();
c = fgetchar();

int c;              return character or code
FILE *fp;           file pointer
```

DESCRIPTION

These functions get a single character from the specified Level 2 file (**stdin** for **fgetchar** and **getchar**). Note that **getc** and **getchar** are actually implemented as macros in order to maximize execution speed.

RETURNS

r = next input character if successful
  = EOF if error or end of file

In the event of an EOF return, error information can be found in **errno** and **_OSERR**. Most programmers treat any EOF return as an indication of end-of-file. However, if you want to distinguish errors from ends-of-file, you should reset **errno** before calling the function and then analyze its contents when you

receive an EOF return.

SEE ALSO

fopen, errno, _OSERR

fgets,gets

Get string from a Level 2 file **Class: ANSI**

NAME

fgets...............Get string from Level 2 file
gets................Get string from stdin

SYNOPSIS

```
#include <stdio.h>

p = fgets(buffer,length,fp);
p = gets(buffer);

char *p;           buffer pointer or NULL
char *buffer;      buffer pointer
int length;        buffer length in bytes
FILE *fp;          file pointer
```

DESCRIPTION

The **fgets** function gets a string from the specified Level 2 file, which must have been previously opened for input. Characters are copied from the file to the buffer until a newline ('\n') has been copied, or the buffer is full, or the end-of-file is hit. In the newline case, a null byte ('\Ø') is placed into the buffer after the newline if the buffer has room. In the end-of-file case, a null byte is placed into the buffer after the last byte that was read. If the end-of-file is hit before any bytes are read, a NULL pointer is returned.

The **gets** function copies characters from **stdin** (the standard input file) until a newline is reached. The newline is not copied to the buffer, but a null byte ('\Ø') is put there in its place.

Get string from a Level 2 file **Class: ANSI**

CAUTIONS

Note that **fgets** will not return a null-terminated string if **length** characters have already been placed into the buffer. Also, make sure that your **gets** buffer can hold the largest line that will be encountered while reading **stdin,** because the function does not have any way to check for a maximum length.

RETURNS

Both functions return the **buffer** argument unless an end-of-file or I/O error occurs, in which case a NULL pointer is returned.

SEE ALSO

errno, fopen, feof, ferror, fgetc, getc

EXAMPLES

```
/*
*
* Assume that stdin contains the following lines:
*
*       Hello, folks!
*       Goodbye, folks!
*
*/
#include <stdio.h>

char *p,b[80];

/* For the next two lines, p will point to b */

p = gets(b);

/*  now b contains "Hello, folks!" */
```

fgets,gets

Get string from a Level 2 file **Class: ANSI**

```
p = fgets(b,sizeof(b),stdio);

/*  now b contains "Goodbye, folks!\n" */

/* After the next line, p is NULL */

p = gets(b);
```

Get file number for L2 file **Class: UNIX**

NAME

fileno..............Get file number for L2 file

SYNOPSIS

```
#include <stdio.h>

fh = fileno(fp);

int fh;           file handle
FILE *fp;         file pointer
```

DESCRIPTION

This function returns the file handle (i.e. the file number) associated with the specified file pointer. The file pointer must be one that was returned by **fopen**, **freopen**, or **fdopen**.

RETURNS

As noted above.

CAUTIONS

This function is implemented as a macro. Also, it does not check that **fp** is a valid file pointer.

fmod,modf

Float modulus operations **Class: ANSI**

NAME

fmod................Compute floating point modulus
modf................Split floating point value

SYNOPSIS

```
#include <math.h>

x = fmod(y,z);
x = modf(y,p);

double x,y,z,*p;
```

DESCRIPTION

The **fmod** function returns **y** if **z** is 0. Otherwise, it returns a value that has the same sign as **y**, is less than **z**, and satisfies the relationship

```
y = (i * z) + x
```

where **i** is an integer. This is, in effect, what the expression

```
x = y % z;
```

would produce if the **%** operation were defined for floating point numbers.

The **modf** function separates the integral and fractional parts of **y** and returns them as two doubles. The function return value is the fractional part, and the integral part is placed in the double pointed to by **p**. Both parts have the same sign as **y**. Note that the fractional part is the number that would be obtained by calling **fmod** in the following way:

Float modulus operations **Class: ANSI**

```
x = fmod(y,1.0);
```

CAUTIONS

Make sure that the second argument of **modf** is a pointer to a double. A common error is to use a pointer to an integer.

EXAMPLES

```
#include <math.h>

double r,ff,fi;

r = fmod(5.7,1.5);        /* r contains 1.2 */

ff = modf(r,&fi);         /* ff contains 0.2 */
                          /* fi contains 1.0 */
```

Change mode of Level 2 file **Class: LATTICE**

NAME

fmode...............Change mode of Level 2 file

SYNOPSIS

```
#include <stdio.h>

fmode(fp,mode);

FILE *fp;        file pointer
int mode;        0 => mode A
                 1 => mode B
```

DESCRIPTION

This function is used to change the translation mode of a Level 2 file that has been opened via **fopen, freopen,** or **fdopen.** In mode A, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. In mode B, all data is transferred with no changes. Mode A also detects the CTRL-Z character (0x1A) as a logical end of file mark.

CAUTIONS

The file pointer is not checked for validity.

SEE ALSO

fopen, freopen, fdopen

Open a Level 2 file **Class: ANSI**

NAME

fopen..............Open a Level 2 file

SYNOPSIS

```
#include <stdio.h>

fp = fopen(name, mode);

FILE *fp;        file pointer
char *name;      file name
char *mode;      access mode
```

DESCRIPTION

This function opens a file for buffered access. The **name** string can be any valid file name and may include a device code and directory path. The **mode** string indicates how the file is to be processed, as follows:

fopen

Open a Level 2 file **Class: ANSI**

MODE	CREATE	TRUNC	READ	WRITE	APPEND	TRANSLATE
"r"	No	No	Yes	No	No	Default
"w"	Yes	Yes	No	Yes	No	Default
"a"	Yes	No	No	No	Yes	Default
"r+"	No	No	Yes	Yes	No	Default
"w+"	Yes	No	Yes	Yes	No	Default
"a+"	Yes	No	Yes	No	Yes	Default
"ra"	No	No	Yes	No	No	Mode A
"wa"	Yes	Yes	No	Yes	No	Mode A
"aa"	Yes	No	No	No	Yes	Mode A
"ra+"	No	No	Yes	Yes	No	Mode A
"wa+"	Yes	No	Yes	Yes	No	Mode A
"aa+"	Yes	No	Yes	No	Yes	Mode A
"rb"	No	No	Yes	No	No	Mode B
"wb"	Yes	Yes	No	Yes	No	Mode B
"ab"	Yes	No	No	No	Yes	Mode B
"rb+"	No	No	Yes	Yes	No	Mode B
"wb+"	Yes	No	Yes	Yes	No	Mode B
"ab+"	Yes	No	Yes	No	Yes	Mode B

CREATE -- Yes
The file will be created if it does not already exist.

CREATE -- No
The function will fail if the file does not already exist.

TRUNC -- Yes
If the file exists, it will be truncated (i.e.

marked as empty).

TRUNC -- No
If the file exists, its current contents will not be disturbed.

READ -- Yes
The file can be read via functions such as **fread** and **fgetc**. Also, **fseek** can be used to position the file before reading.

READ -- No
The file cannot be read.

WRITE -- Yes
The file can be written via functions such as **fwrite** and **fputc**. Also, **fseek** can be used to positon the file before writing.

WRITE -- No
The file cannot be written, but see APPEND below.

APPEND -- Yes
The file can be written, but it is automatically positioned to the current end-of-file before each write operation. This effectively prevents existing data from being changed.

APPEND -- No
Automatic positioning to the end-of-file is not done before a write operation. Also, writes are not allowed unless WRITE is "Yes".

TRANSLATE -- Default
The external integer **_fmode** is used to set mode A or mode B as follows:

```
if(_fmode & 0x8000) set mode B
else set mode A
```

For AmigaDOS, **_fmode** is normally 0x8000.

TRANSLATE -- Mode A
On a read operation, each carriage return character ('\r') is deleted, and the CTRL-Z character is treated as a logical end-of-file mark. On a write operation, each line feed character ('\n') is expanded to a carriage return followed by a line feed.

TRANSLATE -- Mode B
The data is unchanged as it is read or written.

If the file is successfully opened, the function returns a pointer to a "buffered I/O control block", which is defined in the header file **stdio.h.** Normally you will not need to access any information in the control block directly, but you should be very careful not to disturb the block accidentally. A common C programming error is to accidentally mutilate one of these control blocks, which can cause garbage to be written into a file.

RETURNS

A NULL pointer is returned if the file cannot be opened. Consult **errno** and **_OSERR** for detailed error information.

SEE ALSO

fclose, fdopen, fgetc, fgets, fputc, fputs, fread, freopen, fwrite

Create a child process **Class: LATTICE**

NAME

```
forkl...............Fork with arg list
forkv...............Fork with arg vector
wait................Wait for child process to complete
waitm...............Wait for multiple child processes
```

SYNOPSIS

```
#include <dos.h>

error = forkl(prog,arg0,arg1,...,argn,NULL,env,procid);
error = forkv(prog,argv,env,procid);

cc = wait(procid);
complist = waitm(proclist);

int error;                  error code
char *prog;                 program name
char *arg0;                 argument #0
char *arg1;                 argument #1
char *argn;                 argument #n
char *argv[];               argument vector
struct FORKENV *env;        pointer to pseudo environment
                            structure (may be NULL)
struct ProcID *procid;      pointer to child process ID
                            structure
struct ProcID **proclist;   address of pointer to linked
                            list of process ID's
struct ProcID *complist;    pointer to a linked list of
                            completed process ID's
```

DESCRIPTION

These functions create a "child process" by loading a new program as a concurrent process. When the child process completes, the current program (i.e. the "parent process") can obtain its completion code via

the **wait** or **waitm** function. The parent and child are multiprogrammed; that is, the parent continues to execute until it calls either the **wait** or **waitm** function.

You can specify the arguments for the new program in two ways. For the "list method", the function call includes a list of argument string pointers terminated by a NULL pointer. For the "vector method", the function call includes a single pointer to an array of argument string pointers, with the array being terminated by a NULL pointer. Following UNIX conventions, the first argument (i.e. **arg0** or **argv[0]**) should be the program name and is normally the same as **prog.** Under AmigaDOS, the arguments are all concatenated into a pseudo-command line, with a blank separating adjacent arguments and a carriage return character at the end. The maximum size of this line is 255 bytes.

The pseudo environment pointer, if specified, contains optional data pertaining to default files and process execution as follows:

```
struct FORKENV {
        long priority;            /* new process priority          */
        long stack;               /* stack size for new process    */
        long stdin;               /* stdin file handle for new process  */
        long stdout;              /* stdout file handle for new process */
        long console;             /* console window for new process     */
        struct MsgPort *msgport;  /* msg port to receive termination    */
        };                        /* message from child                 */
```

The child process will be supplied with default values for any field left NULL. In addition, a NULL pointer may be passed for this parameter, which will cause default values to be used for all items. If the default values are used, the child process is created

Create a child process **Class: LATTICE**

with a priority of 5, a stack size of 4000 bytes, the current stdin, stdout, and console for the parent process, and a new message port is created to receive the termination message.

The new process executes as a CLI type task. This means it will be expecting file handles for stdin and stdout to be present, and a console task handler to exist. If the parent process is running as a Workbench process then it is possible for none of these to exist for the child process to inherit. If it will be possible for the parent process to be invoked from Workbench then extra effort should be made to ensure the presence of file handles the child process may require.

The optional message port field is provided to allow the parent process to detect when a child process has terminated while awaiting other events, such as Intuition menu events. The termination message format is similar to an Intuition message.

```
struct TermMsg {                /* termination message from child */
        struct Message msg;
        long class;             /* class == 0                  */
        short type;             /* message type == 0           */
        struct Process *process; /* process ID of sending task */
        long ret;               /* return value                */
        };
```

When a termination message is received the parent process must still call the **wait** function to remove the message and unload the child process. If the message was removed from the port it must be replaced by calling **PutMsg** before calling **wait**.

The **forkl** and **forkv** functions load the program from

the current path using the AmigaDOS search procedure. Under Workbench version 1.1 only the current directory and the C: directory are searched for the program. Under Workbench 1.2 or later, directories or devices that have been added to the path may also searched. Alternatively, a path specification can be included as part of the program name.

If the specified program file cannot be found, a -1 return is made, and additional error information can be found in **errno** and **_OSERR.** Upon succesful creation of the child process, the **fork** and **forkl** functions fill in the process ID structure. The address of this structure must be passed as the last argument. The process ID structure is defined in **dos.h** and has the following format:

```
struct ProcID {                        /* packet returned from fork()  */
        struct ProcID *nextID;         /* link to next packet          */
        struct Process *process;       /* process ID of child          */
        int UserPortFlag;              /* used by wait()               */
        struct MsgPort *parent;        /* termination msg destination  */
        struct MsgPort *child;         /* child process' task msg port */
        long seglist;                  /* child process' segment list  */
        };
```

The **nextID** field may be used to link process ID structures into a simple linked list for use with the **waitm** function. The **process** field is the address of the process's task structure, and is the value used with ROM Kernel functions such as **Signal** that require a task ID parameter. The remaining fields are used by the **wait** functions.

The **wait** or **waitm** function must be called for each child process to ensure that the child process terminates cleanly. The **wait** function takes as its single argument a pointer to the ProcID structure for the child task. It waits for a termination message from one particular process, replying to and discarding any other messages that arrive at the message port. The function returns the child process's completion code. This is the value that was passed to the **exit** function when the child terminated.

If any child processes share a message port, however, then **waitm** should be called to ensure that no termination messages are lost. The **waitm** function requires the address of a pointer to the first ProcID structure in a linked list of child process ProcID structures. It waits for one or more termination messages, removing the ProcID structure from the original linked list and inserting it into a linked list of terminated process ProcID structures. The completion code is placed in the UserPortFlag field of the structure. The function then returns a pointer to the first structure in the list of terminated process structures. Since the original list is updated, it may be reused to wait for the remaining child processes.

CAUTIONS

The **wait** or **waitm** function **MUST** be called for each child process before terminating the parent process. Otherwise, the child process will never be unloaded, and there is a high probability that the system will crash.

SEE ALSO

AmigaDOS functions LoadSeg and CreateProc, exit, wait

EXAMPLES

```
/**
*
* This program creates a child process and displays the
* return code
* The child program name and arguments are taken from
* the command line
*
*/
#include "dos.h"
#include "stdio.h"

struct ProcID child;

void main(argc,argv)
int argc;
char *argv[];
{
int ret;
if (argc < 2)
        {
        printf("no program specified\n");
        printf("usage: fork program [arg1] [arg2] ... [argn]\n");
        exit(0);
        }
printf("parent: beginning fork of %s\n",argv[1]);
if (forkv(argv[1],&argv[1],NULL,&child) == -1)
        {
        printf("error forking child\n");
        exit(0);
        }
else    ret = wait(&child);
printf("parent: %s finished, ret = %d\n",argv[1],ret);
```

```
}

/*
* This example forks multiple child processes
*/

#include "dos.h"

char *child1_argv[] =   "task1",        /* program name */
                        "argument1",    /* 1st argument */
                        "argument2",    /* etc., etc.   */
                        NULL;

char *child2_argv[] =   "task2",        /* program name */
                        "argument1",    /* 1st argument */
                        "argument2",    /* etc., etc.   */
                        NULL;

char *child3_argv[] =   "task3",        /* program name */
                        "argument1",    /* 1st argument */
                        "argument2",    /* etc., etc.   */
                        NULL;
void main()
{
struct ProcID *children, *terminated, *task;
struct ProcID child1, child2, child3;
int taskno;

forkv(child1_argv[0],child1_argv,NULL,&child1);
forkv(child2_argv[0],child2_argv,NULL,&child2);
forkv(child3_argv[0],child3_argv,NULL,&child3);

child2->nextID = &child3;
child1->nextID = &child2;
children = &child1;

while(children)                 /* wait until no more children */
```

```
        {
        terminated = waitm(&children) /* must pass ADDRESS of pointer */
        for (task = terminated; task != NULL; task = task->nextID)
            {
            if (task == &child1) taskno = 1;
            else if (task == &child2) taskno = 2;
            else if (task == &child3) taskno = 3;
            printf("task %d terminated, value = %d\n",
                    taskno,task->UserPortFlag);
            }
        }
    }
```

NAME

```
fprintf.............Formatted print to a file
printf..............Formatted printf to stdout
sprintf.............Formatted print to storage
```

SYNOPSIS

```
#include <stdio.h>

length = fprintf(fp,fmt,arg1,arg2,...);
length = printf(fmt,arg1,arg2,...);
length = sprintf(s,fmt,arg1,arg2,...);

int length;      number of characters generated
char *fmt;       format string
FILE *fp;        file pointer
char *s;         storage pointer
```

See below for arg1, arg2, and so on.

DESCRIPTION

These functions generate a stream of ASCII characters by analyzing the format string and performing various conversion operations on the remaining arguments. The **printf** form sends the output stream to the Level 2 file named **stdout,** which is usually the user's screen (i.e. the "console"). The **fprintf** form is similar to **printf,** but it sends the stream to the Level 2 file specified by **fp.** Finally, the **sprintf** form places the output characters into the storage area whose address is given by **s.** You must ensure that this area is large enough to hold the maximum number of characters that might be generated. Note that **sprintf** also generates a null byte to terminate the stored string.

The **fmt** argument points to a string consisting of

ordinary characters and conversion specifications. The ordinary characters are simply copied to the output, but each conversion specification is replaced by the results of the conversion. These results come from operating sequentially upon the arguments that follow **fmt.** That is, the first conversion specification operates upon **arg1,** the second operates upon **arg2,** and so on. In some cases, as described below, a conversion specification may process more than one argument.

Each conversion specification must begin with a percent character (%). If you want to place an ordinary percent into the output stream, precede it with another percent in the **fmt** string. That is, %% will send a single percent character to the output stream. If the percent is not preceded by a percent, then it introduces a conversion specification, as follows:

```
%[flags][width][.precision][l]type
```

where the brackets [...] indicate optional fields, and the fields have the following definitions:

flags Controls output justification and the printing of signs, blanks, decimal places, and hexadecimal prefixes.

width Specifies the "field width", which is the minimum number of characters to be generated for this format item.

precision Specifies the "field precision", which is the required precision of numeric conversions or the maximum number of characters to be copied from a string, depending on the **type** field.

l The letter l indicates that the argument is of "large size".

type Specifies the type of argument conversion to be done.

FLAGS...

If any flag characters are used, they must appear after the percent and can be any of the following:

Minus (-) This causes the result to be left-adjusted within the field specified by **width** or within the default width.

Plus (+) This flag is used in conjunction with the various numeric conversion types to cause a plus or minus sign to be placed before the result. If it is absent, the sign character is generated only for a negative number.

Blank This flag is similar to the plus, but it causes a leading blank for a positive number and a minus sign for a negative number. If both the plus and the blank flags are present, the plus takes precedence.

Sharp (#) This flag causes special formatting. With the 'o', 'x', and 'X' types, the sharp flag prefixes any non-zero output with Ø, Øx, or ØX, respectively. With the 'f', 'e', and 'E' types, the sharp flag forces the result to contain a

decimal point. With the g and G types, the sharp flag forces the result to contain a decimal point and also prevents the elimination of trailing zeroes.

WIDTH...

The **width** is a non-negative number that specifies the minimum field width. If fewer characters are generated by the conversion operation, the result is padded on the left or right (depending on the minus flag described above). A blank is used as the padding character unless **width** begins with a zero. In that case, zero-padding is performed. Note that **width** specifies the minimum field width, and it will not cause lengthy output to be truncated. Use the **precision** specifier for that purpose.

If you don't want to specify the field width as a constant in the format string, you can code it as an asterisk (*), with or without a leading zero. The asterisk indicates that the width value is an integer in the argument list. See the examples for more information on this technique.

PRECISION...

The meaning of the **precision** item depends on the field type, as follows:

Type c
The precision item is ignored.

Types d, o, u, x, and X
The precision is the minimum number of digits to appear. If fewer digits are generated, leading zeroes are supplied.

Types e, E, and f
The precision is the number of digits to appear after the decimal point. If fewer digits are generated, trailing zeroes are supplied.

Types g and G
The precision is the maximum number of significant digits.

Type s
The precision is the maximum number of characters to be copied from the string.

As with the width item, you can use an asterisk for the precision to indicate that the value should be picked up from the next argument.

CONVERSION TYPE...

The conversion type can be any of the following:

c => **Single character conversion**
The associated argument must be an integer. The single character in the rightmost byte of the integer is copied to the output.

d => **Decimal integer conversion**
The associated argument must be an integer, and the result is a string of digit characters preceded by a sign. If the plus and blank flags are absent, the sign is produced only for a negative integer. If the "large size" modifier

is present, the argument is taken as a long integer.

e => Double conversion "-d.dde-ddd"
The associated argument must be a double, and the result has the form

```
-d.dde-ddd
```

where **d** is a single decimal digit, **dd** is one or more digits, and **ddd** is an exponent of exactly three digits. The first minus sign is omitted if the floating point number is positive, and the second minus sign is omitted if the exponent is positive. The plus and blank flags dictate whether there will be a sign character emitted if the number is positive. The "large size" modifier is ignored.

E => Double conversion "-d.ddE-ddd"
This is exactly the same as type **e** except that the result has the form

```
-d.ddE-ddd
```

f => Double conversion "-dd.dd"
The associated argument must be a double, and the result has the form

```
-dd.dd
```

where **dd** indicates one or more decimal digits. The minus sign is omitted if the number is positive, but a sign character will still be generated if the plus or blank flag is present. The number of digits before the decimal point

depends on the magnitude of the number, and the number after the decimal point depends on the requested precision. If no precision is specified, the default is six decimal places. If the precision is specified as Ø, or if there are no non-zero digits to the right of the decimal point, then the decimal point is omitted.

g => **Double conversion, general form**
The associated argument must be a double, and the result is in the **e** or **f** format, depending on which gives the most compact result. The **e** format is used only when the exponent is less than -4 or greater than the specified or default precision. Trailing zeroes are eliminated, and the decimal point appears only if any non-zero digits follow it.

G => **Double conversion, general form**
This is identical to the **g** format, except that the **E** type is used instead of **e**.

o => **Octal integer conversion**
The associated argument is taken as an unsigned integer, and it is converted to a string of octal digits. If the "large size" modifier is present, the argument must be a long integer.

p => **Pointer conversion**
The associated argument is taken as a data pointer, and it is converted to hexadecimal representation. Under AmigaDOS, the pointer is printed as 8 hexadecimal digits, with leading zeroes if necessary.

P => Pointer conversion
This is the same as the **p** format, except that upper case letters are used as hexadecimal digits.

s => String conversion
The associated argument must point to a null-terminated character string. The string is copied to the output, but the null byte is not copied.

u => Unsigned decimal integer conversion
The associated argument is taken as an unsigned integer, and it is converted to a string of decimal digits. If the "large size" modifier is present, the argument must be a long integer.

x => Hexadecimal integer conversion
The associated argument is take as an unsigned integer, and it is converted to a string of hexadecimal digits with lower case letters. If the "large size" modifier is present, the argument is taken as a long integer.

X => Hexadecimal integer conversion
This is the same as the **x** format, except that upper case letters are used as hexadecimal digits.

RETURNS

Each function returns the number of output characters generated. For **sprintf,** this number does not include the terminating null byte.

Formatted print **Class: ANSI**

EXAMPLES

```
/*
*
* This example prints a message indicated whether
* the function argument is positive or negative.
* In the second "printf", the width and precision
* are 15 and 8, respectively.
*
*/
#include <stdio.h>

pneg(value)
double value;
{
char *sign;

if(value < 0) sign = "negative";
else sign = "not negative";

printf("The number %E is %s.\n",value,sign);

printf("The number %*.*E is %s.\n",value,15,8,sign);
}
```

fputc,putc,putchar,fputchar

Put a character **Class: ANSI**

NAME

fputc...............Put a character to a level 2 file
putc................Put a character to a level 2 file
putchar.............Put a character to stdout
fputchar............Put a character to stdout

SYNOPSIS

```
#include <stdio.h>

r = fputc(c,fp);
r = putc(c,fp);
r = putchar(c);
r = fputchar(c);

int r;          EOF or c
int c;          Character to be output
FILE *fp;       Level 2 file pointer
```

DESCRIPTION

These functions put a single character to the specified Level 2 file (**stdout** for **fputchar** and **putchar**). Note that **putc** and **putchar** are actually implemented as macros in order to maximize execution speed.

RETURNS

r = c if successful
  = EOF if error

For disk files, an EOF return usually means that the disk is full. However, this type of return can also occur if the device is write-protected or if a write error occurs. In any case, additional error informa-

tion can be found in **errno** and **_OSERR**.

SEE ALSO

fopen, errno, _OSERR

fputs,puts

Put string to Level 2 file **Class: ANSI**

NAME

fputs...............Put string to Level 2 file
puts................Put string to stdout

SYNOPSIS

```
#include <stdio.h>

error = fputs(s,fp);
error = puts(s);

int error;        non-zero if error
char *s;          string pointer
FILE *fp;         file pointer
```

DESCRIPTION

The **fputs** function copies string **s** to a level 2 file that was previously opened for output. The string must be terminated by a null byte, which is not copied.

The **gets** function copies string **s** to **stdout** (the standard output file). The terminating null byte is not copied, but a newline is sent after the string.

RETURNS

If an error occurs, the return value is -1; otherwise, it is 0. Additional error information can be found in **errno** and **_OSERR**.

SEE ALSO

errno, ferror, fopen, fputc

Put string to Level 2 file **Class: ANSI**

EXAMPLES

```
/*
*
* This examples writes the following two lines to stdout:
*
*       This is the first line
*       This is the second line
*
*/
#include <stdio.h>

puts("This is the first line");
fputs("This is ",stdout);
puts("the second line");
```

Read and write blocks **Class: ANSI**

NAME

fread..............Read blocks from a Level 2 file
fwrite.............Write blocks to a Level 2 file

SYNOPSIS

```
#include <stdio.h>

a = fread(b,bsize,n,fp);
a = fwrite(b,bsize,n,fp);

int a;          actual number of blocks
char *b;        pointer to first block
int bsize;      size of block in bytes
int n;          maximum number of blocks
FILE *fp;       file pointer
```

DESCRIPTION

These functions perform Level 2 I/O operations to read and write blocks of data. Each block contains **bsize** bytes and up to **n** blocks are stored into contiguous memory locations beginning at location **b**.

For **fread,** blocks are read until **n** have been stored or until the end-of-file is hit. If the end-of-file is hit in the middle of a block, that partial block will be stored in the **b** array, but it will not be included in the function return value. In other words, the return value indicates the number of complete blocks that were read.

For **fwrite,** blocks are written until **n** have been sent or until the output device cannot accept any more. If the output device becomes full in the middle of a block, a partial block will be written, but it will not be included in the function return value. In other

words, the return value indicates the number of complete blocks that were written.

RETURNS

The functions return the number of complete blocks that were processed.

SEE ALSO

fopen, fclose, ferror, feof, fgetc, fputc, fseek

freopen

Reopen a Level 2 file **Class: ANSI**

NAME

freopen............Reopen a Level 2 file

SYNOPSIS

```
#include <stdio.h>

fpr = freopen(name, mode, fp);
FILE *fpr;          file pointer after re-opening
char *name;         file name
char *mode;         access mode
FILE *fp;           current file pointer
```

DESCRIPTION

This function reopens a Level 2 file. That is, it attaches a new file to a previously used file pointer. The previous file is automatically closed before the file pointer is reused. The name and mode arguments are the same as those for **fopen**.

RETURNS

fpr = NULL if error
    = fp if successful

CAUTIONS

The return code should be checked for NULL; the same errors as defined for "fopen" may occur. Also, for complete portability, do not assume that **fpr** and **fp** are identical. Use **fpr** to access the reopened file, not **fp**.

SEE ALSO

fopen, fdopen

Split or combine float value **Class: ANSI**

NAME

frexp...............Split fraction and exponent
ldexp...............Load exponent

SYNOPSIS

```
#include <math.h>

f = frexp(v,xp);
v = ldexp(m,x);

double f;        fraction
double v;        value
int *xp;         exponent pointer
int x;           exponent
```

DESCRIPTION

The **frexp** function splits floating point value **v** into its fraction (mantissa) and exponent parts. The mantissa is returned as a double whose absolute value is greater than or equal to 0.5 and less than 1.0. The exponent is returned as an integer whose absolute value is less than 1024.

The **ldexp** function adds the integer **x** to the exponent in **f**, which is the same as computing

```
v = f * (2 ** x)
```

Note that if **f** and **x** are the results of **frexp**, then **ldexp** performs the reverse operation. Also, if the absolute value of the resulting exponent is greater than 1023, then **matherr** will be called with an overflow or underflow error indication.

frexp,ldexp

Split or combine float value **Class: ANSI**

SEE ALSO

fmod, matherr, modf

NAME

fscanf.............Formatted input from a file
scanf..............Formatted input from stdin
sscanf.............Formatted input from a string

SYNOPSIS

```
#include <stdio.h>

n = fscanf(fp,fmt,arg1,arg2,...);
n = scanf(fmt,arg1,arg2,...);
n = sscanf(ss,fmt,arg1,arg2,...);

int n;          number of input items matched, or EOF
FILE *fp;       file pointer (fscanf only)
char *ss;       input string (sscanf only)
char *fmt;      format string
---- *argx;     pointers to input data areas
```

DESCRIPTION

These functions perform formatted input conversions on text obtained from the standard input file, a specified Level 2 file, or a string. The input characters are read and checked against the format string, which may contain any of the following:

White space
Any number of spaces, horizontal tabs, or newline characters will cause input to be read up to the next character that is not white space.

Ordinary characters
Any character that is not white space and is not the percent sign (%) must match the next input character. Use a double percent (%%) in

the format string to match a single percent in the input. If there is not an exact match, scanning stops, and the function returns.

Conversion specification

This is multi-character sequence that indicates how the next input characters are to be converted. The form is:

```
%*nlt
```

where the various fields are defined as follows:

% A percent sign introduces a conversion specifier. If you want to match a percent sign in the input, indicate this by a double percent (%%) in the format string.

***** The asterisk is optional. If present, it means that the conversion should be performed, but the result should not be stored. There should be no value pointer in the argument list for a suppressed conversion.

n This is an optional decimal number that specifies the maximum input field width. This is used only with the **s** format.

l The letter l is optional. If present, it indicates that a long conversion should be performed.

t The **t** stands for one of the following format characters: c, d, e, f, g, i, n,

o, s, u, x. These are described below.

If the conversion is successful and assignment is not suppressed, the result is placed into the corresponding argument. The argument list must contain a pointer to an appropriate data item for each conversion specification that does not suppress assignment.

The function returns the number of conversion values that were assigned. This can be less than the number expected if the input characters do not agree with the format string. If an end-of-input is reached before any values are assigned, the return value is EOF.

The format characters listed above specify how the input characters are to be converted. Leading white space is skipped in all cases except the **c** conversion.

c => **character**
The corresponding argument must point to a character. The next input character is moved to that destination. No white space is skipped.

d => decimal number
The corresponding argument must point to an integer or to a long integer. The latter applies if the **d** is preceded by an **l**. The input characters should be decimal digits, optionally preceded by a plus or minus sign.

e,f,g => floating point
These three types are identical. The corresponding argument must point to a float or a double. The latter applies if the type letter is preceded by an **l**. The input characters must consist of the following sequence:

1. Optional leading white space.

2. An optional plus (+) or minus (-).

3. A sequence of decimal digits.

4. An optional decimal point followed by 0 or more decimal digits.

5. An optional exponent, consisting of the letter **e** or **E** followed by an optional plus or minus sign followed by 1 or more decimal digits. This general form is shown below, where [...] indicates an optional part:

```
[whitespace][sign]digits[.digits][exponent]
```

n => character count
No input characters are read. The corresponding argument must point to an integer into which is written the number of input characters read so far.

o => octal number
An octal number is expected, and the corresponding argument should point to an integer, or to a long integer if the **o** is preceded by an **l**.

s => string
A character string is expected, and the corresponding argument should point to a character array large enough to hold the string and a terminating null byte. The input string is

terminated by white space or the end-of-input. Also, if a maximum field width is specified, the output array size should be at least that width plus 1, because the reading of input characters will stop at the field width even if no white space has been hit.

u => unsigned number
An unsigned decimal number is expected, and the corresponding argument should point to an unsigned integer, or to an unsigned long integer if the **u** is preceded by an **l**.

x => hexadecimal number
A hexadecimal number is expected, and the corresponding argument should point to an integer, or to a long integer if the **x** is preceded by an **l**. The hexadecimal number can begin with the characters "0x" or "0X", and case is not significant for the hexadecimal letters.

RETURNS

The function returns the number of assignments that were made. For example, a return value of 3 indicates that conversion results were assigned to **arg1**, **arg2**, and **arg3**.

CAUTIONS

All of the result arguments (i.e. arg1, arg2, and so on) must be pointers. Also, you should not supply a pointer for any conversion specification that uses the * to suppress assignment.

fseek,ftell,rewind

Set or get L2 file position **Class: ANSI**

NAME

fseek...............Set Level 2 file position
ftell...............Get Level 2 file position
rewind..............Seek to beginning of Level 2 file

SYNOPSIS

```
#include <stdio.h>

error = fseek(fp,rpos,mode);
apos = ftell(fp);
error = rewind(fp);

int error;        non-zero if error
FILE *fp;         file pointer
long rpos;        relative file position
int mode;         seek mode
long apos;        absolute file position
```

DESCRIPTION

The **fseek** function moves the byte cursor of a Level 2 file to a new position. The **mode** argument must be one of the following:

0 The **rpos** argument is the number of bytes from the beginning of the file. This value must be positive.

1 The **rpos** argument is the number of bytes relative to the current position. This value can be positive or negative.

2 The **rpos** argument is the number of bytes relative to the end of the file. This value must be negative or zero.

Set or get L2 file position **Class: ANSI**

The **rewind** function resets the specified file to its first byte and is equivalent to the following **fseek** call:

```
error = fseek(fp,0L,0);
```

In fact, **rewind** is implemented as a macro that simply calls **fseek** in that way.

The **ftell** function returns a long value that is the current byte position in the file, relative to the beginning. It is equivalent to the following call:

```
apos = lseek(fp->_file,0L,1);
```

and it is implemented as a true function, not as a macro.

RETURNS

For **fseek,** a value of -1 is returned if an error occurs, and for **ftell** an error is indicated by a return value of -1L. In either case, **errno** and _OSERR contain additional error information.

SEE ALSO

fopen, errno, _OSERR, lseek, tell

gcvt

Convert float to string **Class: UNIX**

NAME

gcvt................Convert float to string

SYNOPSIS

```
#include <math.h>

p = gcvt(v,dig,buffer);

char *p;          points to buffer
double v;         floating point value
int dig;          number of significant digits
char *buffer;     output buffer
```

DESCRIPTION

This function converts the specified floating point value into a null-terminated string in the output buffer. The string will be in either of two formats. First **gcvt** attempts to produce **dig** significant digits in the FORTRAN F format. If that fails, it produces **dig** significant digits in the FORTRAN E format. Trailing zeroes will be eliminated if necessary.

RETURNS

The function returns a pointer to the buffer.

CAUTIONS

Make sure that the buffer is large enough.

SEE ALSO

ecvt, fcvt

Convert float to string Class: UNIX

EXAMPLES

```
/*
*
* This example displays 314150.0
*
*/
#include <stdlib.h>

char s[100];

printf("%s\n",gcvt(-3.1415e5,7,s));
```

getcd

Get current directory **Class: AMIGA**

NAME

getcd...............Get current directory

SYNOPSIS

```
#include <dos.h>

error = getcd(drive,path);

int error;        0 if successful
int drive;        drive code
char *path;       points to 64-byte path area
```

DESCRIPTION

This function gets the current directory path for the specified disk drive. The drive specification is retained in this implementation for compatibility with other implementations. However, only a value of Ø (indicating the current drive) is supported under AmigaDOS. Any other value is considered an error.

Note that the path area must be large enough to contain the expected path. The returned string will contain the entire path, including the volume name of the device.

RETURNS

If the operation is successful, the function returns Ø. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

getcwd, errno, _OSERR

Get or change system clock **Class: AMIGA**

NAME

getclk..............Get system clock
chgclk..............Change system clock

SYNOPSIS

```
#include <dos.h>

getclk(clock);
error = chgclk(clock);

int error;
unsigned char *clock;
```

DESCRIPTION

The **getclk** function obtains the current setting of the system clock and places it into an 8-byte array as follows:

```
clock[0] => day of week (0 for Sunday)
clock[1] => year - 1980
clock[2] => month (1 to 12)
clock[3] => day (1 to 31)
clock[4] => hour (0 to 23)
clock[5] => minute (0 to 59)
clock[6] => second (0 to 59)
clock[7] => hundredths (0 to 99)
```

The **chgclk** function changes the setting of the system clock to the value specified in the array.

RETURNS

If the array is invalid or the operation failed, **chgclk** returns a non-zero value.

Get or change system clock **Class: AMIGA**

CAUTIONS

If your machine is equipped with a hardware clock, its state is not necessarily changed by a call to **chgclk.**

SEE ALSO

errno, _OSERR

Get current working directory **Class: UNIX**

NAME

getcwd.............Get current working directory

SYNOPSIS

```
#include <stdio.h>

p = getcwd(b,size);

char *p;            Same as b is successful, else NULL
char *b;            Points to path buffer
int size;           Size of path buffer
```

DESCRIPTION

This function obtains the path name for the current working directory. If the buffer pointer **b** is not null, then the path string is placed there if it will fit, and the return pointer **p** is the same as **b.** If **b** is null, then **malloc** is used to obtain a buffer of **size** bytes to hold the path string. In this latter case, you should use the **free** function to release the buffer when you are finished with it.

Note that the **getcd** function is often more efficient under AmigaDOS since it does not use memory allocation.

RETURNS

If the operation is successful, the function returns a pointer to the buffer. Otherwise it returns a null pointer and places error information in **errno** and **_OSERR.** Also, a null pointer is returned if the path string will not fit in the buffer or if a buffer cannot be allocated. In either of those cases, **errno** is unchanged, and **_OSERR** is reset.

getcwd

Get current working directory **Class: UNIX**

SEE ALSO

getcd, errno, _OSERR

Get disk free space **Class: AMIGA**

NAME

getdfs..............Get disk free space

SYNOPSIS

```
#include <dos.h>

error = getdfs(drive,info);

int error;                  0 if successful
char *drive;                drive or volume name
                            (NULL => current drive)
struct DISKINFO *info;      disk information
```

DESCRIPTION

This function obtains information about the specified disk drive, including the amount of free space available. If a null pointer is passed as the drive name, information is obtained about the current drive. The DISKINFO structure is defined in **dos.h.** For AmigaDOS, this is the same as the AmigaDOS InfoData structure:

```
struct InfoData {
    LONG   id_NumSoftErrors;
    LONG   id_UnitNumber;
    LONG   id_DiskState;
    LONG   id_NumBlocks;
    LONG   id_NumBlocksUsed;
    LONG   id_BytesPerBlock;
    LONG   id_DiskType;
    BPTR   id_VolumeNode;
    LONG   id_InUse;
};
```

getdfs

Get disk free space **Class: AMIGA**

RETURNS

A return value of 0 indicates success. If the drive code is invalid or no disk is mounted on that drive, then the return value is -1. Note that no additional information is provided in **errno** or **_OSERR**.

EXAMPLES

```
/* Compute number of bytes available on current drive: */

#include <dos.h>
struct DISKINFO info;
long size;

if(getdfs(0,&info) == 0)
    size = (info.id_NumBlocks - info.id_NumBlocksUsed)
            * info.id_BytesPerBlock;
```

Get file attribute **Class: AMIGA**

NAME

getfa...............Get file attribute

SYNOPSIS

```
#include <dos.h>

fa = getfa(name);

int fa;          1, -1, or 0
char *name;      file name
```

DESCRIPTION

This function determines whether or not the specified file is a directory. The status is returned in **fa** and contains the following information:

```
 1 => Directory file
 Ø => Error
-1 => Normal file
```

RETURNS

If the operation is unsuccessful, the function returns Ø and places error information in **errno** and **_OSERR.**

SEE ALSO

errno, _OSERR

getfnl

Get file name list **Class: LATTICE**

NAME

getfnl..............Get file name list

SYNOPSIS

```
#include <stdlib.h>

n = getfnl(fnp,fna,fnasize,attr);

int n;                  number of matching file names
char *fnp;              file name pattern
char *fna;              file name array
unsigned fnasize;       size of file name array
int attr;               file attribute
```

DESCRIPTION

This function gets all file names that match the specified pattern and attribute, and it places them into the file name array. Each name is stored as a null-terminated string, and the file name array is terminated by a null string (i.e. a string consisting of only a null byte). If the file name pattern includes a path prefix, that prefix is placed in front of each matching file name.

The function return value is the number of strings stored in the array, not including the terminating null string.

The file name pattern has the general form

```
drive:path/node.ext
```

The function first strips off the drive and directory path portion and restricts its search to that area of the file system. The node and extension parts can

contain any valid file name characters, including the wildcard pattern matching characters. AmigaDOS wildcard characters are supported by default. Setting the external integer location **msflag** to a nonzero value will cause MS-DOS wildcard characters * and ? to be supported instead. Some examples are:

"a:#?.c"
: Finds all files on drive A that have ".c" as their extension. A file named "abc.c" would thus be place in the array as "a:abc.c".

":abc/def/q*.x?"
: Finds all files in the directory **:abc/def** that begin with the letter q and have extensions consisting of the letter x and one other letter. For example, one such name would be "/abc/def/queen.x".

"XYZ*."
: Finds all files in the current directory that begin with "XYZ" and have no extension. One example is "XYZ".

Notice that AmigaDOS makes no distinction between upper and lower case in any part of the file name.

The attribute is an integer defined as follows:

```
Ø => non directory files
1 => all files including directories
```

RETURNS

A value of -1 is returned if the file name pattern is invalid or if there is not enough room in the file

name array. In the first case, **_oserr** will contain further error information.

SEE ALSO

dfind, dnext, strbpl, strsrt, _oserr

EXAMPLES

```
/*
*       This program constructs an array of pointers
*       to all normal files in the current directory
*       that have an extension of ".c".  Then
*       the array is sorted into ASCII order.
*
*/

char names[3000],*pointers[300];
int count;

count = getfnl("#?.c",names,sizeof(names),0);

if(count > 0)
    {
    if(strbpl(pointers,300,names) != count)
        {
        fprintf(stderr,"Too many file names\n");
        exit(1);
        }
    strsrt(pointers,count);
    }
else
    {
    if(_OSERR) poserr("FILES");
    else fprintf(stderr,"Too many files\n");
    exit(1);
    }
```

Get file time **Class: AMIGA**

NAME

getft...............Get file time

SYNOPSIS

```
#include <dos.h>

ft = getft(name);

long ft;          file time or -1 if error;
char *name;       file name
```

DESCRIPTION

This function gets the time and date information associated with the specified file. This information usually indicates when the file was created or last updated. This function returns the file time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970.

RETURNS

If **getft** is successful, the file time (a long integer) is returned. Otherwise a value of -1L is returned. Additional error information can be found in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

getmem,getml

Get Level 2 memory block **Class: LATTICE**

NAME

getmem..............Get Level 2 memory block (short)
getml...............Get Level 2 memory block (long)

SYNOPSIS

```
#include <stdlib.h>

p = getmem(sbytes);
p = getml(lbytes);

char *p;          block pointer
short sbytes;     number of bytes
long lbytes;      number of bytes
```

DESCRIPTION

These functions allocate a block from the Level 2 memory pool and return a pointer to the first byte in the block. If the pool does not currently contain a block of sufficient size, the Level 1 memory allocator, **sbrk,** is called to obtain more space from the operating system. If that step fails, a NULL pointer is returned.

Unlike the Level 3 memory allocator, Level 2 does not allocate any additional bytes to retain the block size and other overhead information. However, Level 2 does require a minimum block size of **MELTSIZE** bytes, where **MELTSIZE** is defined in the **dos.h** header file. If you request a block that is smaller, **getmem** and **getml** will round it up to the minimum size.

RETURNS

A NULL pointer is returned if the block could not be allocated. Otherwise, a character pointer is returned,

but it can be cast to any other pointer type.

CAUTIONS

Since the Level 2 memory allocator does not retain any information about the blocks it has allocated, you must keep track of the block pointer and size if you plan to release the memory at some later time.

SEE ALSO

rlsmem, rlsml, sizmem

EXAMPLES

```
/*
*
* This program builds a linked list of text strings
* obtained from the standard input file.
*
*/
#include <stdio.h>
#include <stdlib.h>
/*
* These elements are linked together to form the
* text string list.
*/
struct LIST
  {
  struct LIST *next;      /* forward linkage */
  int size;               /* element size */
  char text[2];           /* minimum text string */
  };
/*
*
* Main program
*
```

```
*/
main()
{
struct LIST *p,*q,*list = NULL;
char b[256];
int x;
/*
*
* Build the list
*
*/
for(q = (struct LIST *)(&list);;q = p)
  {
  printf("Enter a text string: ");
  if(gets(b) == NULL) break;
  if(b[0] == '\0') break;
  x = sizeof(struct LIST) - 2 + strlen(b) + 1;
  p = (struct LIST *)getmem(x);
  if(p == NULL)
    {
    printf("No more memory\n");
    break;
    }
  p->next = q->next;
  p->size = x;
  strcpy(p->text,b);
  }
 /*
 *
 * Print the list
 *
 */
 printf("\n\nTEXT LIST...\n");
 for(p = list; p != NULL; p = p->next)
    printf("%s",p->text);
 }
```

Unpack Greenwich Mean Time **Class: ANSI**

NAME

gmtime..............Unpack Greenwich Mean Time
localtime...........Unpack local time

SYNOPSIS

```
#include <time.h>

ut = gmtime(t);
ut = localtime(t);

struct tm *ut;
long *t;
```

DESCRIPTION

These functions unpack a time value from the long integer form into a structure. Normally the time value represents the number of seconds since 00:00:00, January 1, 1970, Greenwich Mean Time. The **time** function returns this kind of number. For **gmtime,** this number is converted as is, while **localtime** adjusts the number for the local time zone.

SEE ALSO

asctime, ctime, localtime, time

CAUTIONS

Note that these functions expect a pointer as the argument. A common error is to pass the actual time value instead of the pointer.

Also, the functions share a static data area for their return values, and a call to either one will destroy the results of the previous call.

Unpack Greenwich Mean Time **Class: ANSI**

EXAMPLES

```
#include <time.h>
main()
{
struct tm *p;
long t;

time(&t);
p = gmtime(&t);
printf("GMT is %s\n",asctime(p));
}
```

Character tests **Class: ANSI**

NAME

```
isalnum.............Test if alphanumeric character
isalpha.............Test if alphabetic character
isascii.............Test if ASCII character
iscntrl.............Test if control character
iscsym..............Test if C symbol character
iscsymf.............Test if C symbol lead character
isdigit.............Test if decimal digit character
isgraph.............Test if graphic character
islower.............Test if lower case character
isprint.............Test if printable character
ispunct.............Test if punctuation character
isspace.............Test if space character
isupper.............Test if upper case character
isxdigit............Test if hex digit character
```

SYNOPSIS

```
#include <ctype.h>

t = isalnum(c);
t = isalpha(c);
t = isascii(c);
t = iscntrl(c);
t = iscsym(c);
t = iscsymf(c);
t = isdigit(c);
t = isgraph(c);
t = islower(c);
t = isprint(c);
t = ispunct(c);
t = isspace(c);
t = isupper(c);
t = isxdigit(c);

int t;          truth value (0 if false, non-zero if true)
int c:          character to test
```

Character tests　　　　　　　　　　　　Class: ANSI

DESCRIPTION

These functions test for various character types. If you include **ctype.h** as shown above, then the functions are actually defined as macros and generate in-line code to test the static array named **_ctype.** This array contains a bit mask for each of the 256 possible character values and for the integer value -1. See the **ctype.h** description for an explanation of this array.

If you don't include **ctype.h,** these functions will be resolved in the library, which can reduce your program size slightly at the expense of execution speed. If you want to use the function versions but must include **ctype.h** for some other reason, use **#undef** to undefine the appropriate character test macros.

CAUTIONS

You can use either characters or integers as arguments, but the macros are defined only over the integer range from -1 to 255. The functions, however, will correctly handle the entire integer range.

The reason -1 is included as a valid argument is to avoid a nonsense result if you feed the EOF value to one of the macros or functions. EOF can be returned by **getchar** and other I/O functions, and if you pass it to any of the character test functions, the resulting truth value will be zero.

SEE ALSO

_ctype

EXAMPLES

```
#include <stdio.h>
#include <ctype.h>
main()
char b[100];
int c;

while((c = getchar()) != EOF)
        printf("\n%c %s alphabetic.\n",c,
                isalph(c) ? "is" : "is not");
}
```

Perform long jump **Class: ANSI**

NAME

longjmp.............Perform long jump
setjmp..............Set long jump parameters

SYNOPSIS

```
#include <setjmp.h>

ret = setjmp(save);
longjmp(save,value);

int ret;          return code
int value;        return value
jmp_buf *save;    address of save area
```

DESCRIPTION

The **setjmp** function checkpoints the current stack mark in the save area and returns a code of 0. A subsequent call to **longjmp** will then cause control to return to the next statement after the original **setjmp** call, with **value** as the return code. If **value** is 0, it is forced to 1 by **longjmp**.

This mechanism is useful for quickly popping back up through multiple layers of function calls under exceptional circumstances. Structured programming gurus lose a lot of sleep over the "pathological connections" that can result from indiscriminate usage of these functions.

RETURNS

A return code of 0 from **setjmp** indicates that this is the initial call to save the stack. A non-zero return code indicates that **longjmp** has been executed.

Perform long jump **Class: ANSI**

CAUTIONS

Calling **longjmp** with an invalid save area is an effective way to disrupt the system. One common error is to use **longjmp** after the function calling **setjmp** has returned to its caller. This cannot possibly succeed, since the stack frame for that function no longer exists.

lsbrk,rbrk,sbrk

Level 1 memory allocation **Class: UNIX**

NAME

lsbrk...............Allocate Level 1 memory (long)
rbrk................Release Level 1 memory
sbrk................Allocate Level 1 memory (short)

SYNOPSIS

```
#include <stdlib.h>

p = lsbrk(lbytes);
error = rbrk();
p = sbrk(sbytes);

char *p;          block pointer
long lbytes;      number of bytes
short sbytes;     number of bytes
int error;        0 if successful
```

DESCRIPTION

These functions form Level 1 of Lattice's layered memory allocation system. This level is **incompatible** with most versions of UNIX. The memory pool is viewed as a group of non-contiguous areas of memory allocated from the system free memory pool.

The **sbrk** and **lsbrk** functions requests **sbytes or lbytes** of memory from the system, adding the the allocated block to a linked list of memory blocks to be returned to the system when the program terminates. The function returns the address of the block just allocated.

The **rbrk** function returns all memory from this area to the operating system.

CAUTIONS

Calling **rbrk** will free all memory in the memory pool, including level-2 I/O buffers.

RETURNS

If **sbrk** fails, it returns value **(char *)(-1)** which is -1 cast into a character pointer. This strange return is a legacy of UNIX. For **lsbrk,** an error is indicated by a NULL pointer.

SEE ALSO

getmem, malloc

lseek,tell

Set or get file position **Class: UNIX**

NAME

lseek...............Set Level 1 file position
tell................Get Level 1 file position

SYNOPSIS

```
#include <fcntl.h>

apos = lseek(fh,rpos,mode);
apos = tell(fh);

int fh;           file handle
long rpos;        relative file position
int mode;         seek mode
long apos;        absolute file position
```

DESCRIPTION

The **lseek** function moves the byte cursor of a Level 1 file to a new position. The **mode** argument must be one of the following:

0 The **rpos** argument is the number of bytes from the beginning of the file. This value must be positive.

1 The **rpos** argument is the number of bytes relative to the current position. This value can be positive or negative.

2 The **rpos** argument is the number of bytes relative to the end of the file. This value must be negative or zero.

If **lseek** is asked to move 0 bytes relative to the current position, it simply returns the current file position. The **tell** function, then, is equivalent to

Set or get file position **Class: UNIX**

```
apos = lseek(fh,0L,1);
```

RETURNS

Both functions return -1L if an error occurs, in which case **errno** and **_OSERR** contain additional information.

SEE ALSO

MSDOS function Øx42, errno, _OSERR, open

EXAMPLES

```
/**
*
* This program totals the number of bytes used by
* all normal files in the current directory.
*
**/
#include <fcntl.h>                /* for Level 1 I/O */

char names[8192];                 /* holds file names */

main()
{
char *p;
int f,n;
long x,y;

if(getfnl("*.*",names,sizeof(names),0) <= 0)
  {
  printf("Can't build file name list\n");
  exit(1);
  }
for(x = 0, n = 0, p = names; *p != '\0'; p += strlen(p) + 1)
```

```
    {
    f = open(p,O_RDONLY);
    if(f < 0)
      {
      printf("Can't open \"%s\"\n",p);
      exit(1);
      }
    y = lseek(f,0L,2);
    if(y < 0)
      {
      printf("Seek failure on \"%s\"\n",p);
      exit(1);
      }
    x += y;
    n++;
    close(f);
    }
  printf("%d files, %ld bytes used\n",n,x);
  }
```

Your main program **Class: ANSI**

NAME

main................Your main program

SYNOPSIS

```
#include "workbench/startup.h"

void main(argc,argv);

int argc;                  argument count
union {
  char *args[];
  struct WBStartup *msg;
      } argv;              argument vector
```

DESCRIPTION

This function does not actually exist in the library; you must supply one of these "main programs" in each of your applications. If you trace through the two startup modules C.A and _MAIN.C, you will find that C.A passes control to _MAIN.C, which then calls the function named **main.** Since we supply the source code for both of these modules, you are free to change this initialization procedure for special applications. The standard version simulates UNIX's interface with C programs by setting up a "vector", which is simply an array of pointers.

The **argv.args** array contains pointers to the command line arguments, and **argc** indicates how many pointers are in the array. For example, if you invoke **myprog** with the following command line

```
myprog abc def "ghi jkl"
```

then **argv.args** is set up as follows:

```
argv.args[0] => "myprog"
argv.args[1] => "abc"
argv.args[2] => "def"
argv.args[3] => "ghi jkl"
```

and **argc** contains the value 4.

Under WorkBench there is no command line. In this case, **argc** is Ø indicating no command or arguments, and **argv.msg** is a pointer to the WorkBench startup message structure.

RETURNS

None. When **main** returns to its caller (normally _MAIN.C), the program exits to AmigaDOS with a termination code of Ø. If you want to pass a non-zero termination code back to AmigaDOS, use the **exit** or **_exit** function.

SEE ALSO

exit, _exit

EXAMPLES

```
/**
*
* This program is intended to run only under CLI,
* and displays the command and any arguments
*
**/
void main(argc, argv)
int argc;
char *argv[];
{
```

Your main program **Class: ANSI**

```
int i;

printf("command = %s\n",argv[0]);
for (i = 0; argc > 1; i++, argc--)
        printf("argument %d = %s\n",i,argv[i]);
}

/**
*
* This program is intended to run only under WorkBench,
* and gets its arguments from the WorkBench message structure
*
**/
#include "workbench/startup.h"

void main(argc, argv)
int argc;
struct WBStartup *argv;
{
int i;

if (argc != 0) exit(ERROR);

printf("command = %s\n",argv->sm_ArgList[0]->wa_name);
for (i = 1; i < argv->sm_NumArgs; i++)
 printf("argument %d = %s\n",i,argv->sm_ArgList[i]->wa_name);
}

/**
*
* This program is intended to run under either
* WorkBench or CLI.
*
**/
#include "workbench/startup.h"

void main(argc, argv)
```

Your main program **Class: ANSI**

```
int argc;
union  {
        char *args[];
        struct WBStartup *msg;
        } argv;         argument vector
{
int i;

if (argc != 0)
  {
  printf("command = %s\n",argv.args[0]);
  for (i = 0; argc > 1; i++, argc--)
      printf("argument %d = %s\n",i,argv.args[i]);
  }
else
  {
  printf("command = %s\n",
          argv.msg->sm_ArgList[0]->wa_name);
  for (i = 1; i < argv.msg->sm_NumArgs; i++)
      printf("argument %d = %s\n",i,
              argv.msg->sm_ArgList[i]->wa_name);
   }
}
```

NAME

matherr.............Math error handler
except..............Call math error handler

SYNOPSIS

```
#include <math.h>

a = matherr(x);
r = except(type,name,arg1,arg2,retval);

int a;                     action code
struct exception *x;       exception vector
double r;                  actual return value
int type;                  error type
char *name;                math function name
double arg1;               first argument
double arg2;               second argument
double retval;             proposed return value
```

DESCRIPTION

The **matherr** function is called whenever one of the higher-level math functions detects an error. The exception vector structure is defined in **math.h** and contains information about the error as follows:

```
struct exception
  {
  int type;               /* error type */
  char *name;             /* math function name */
  double arg1, arg2;      /* function arguments */
  double retval;          /* proposed return value */
  };
```

The standard library version of **matherr** translates the error type into a UNIX error code that is placed into

errno. Then the function returns an action code of 0 to indicate that the math function should simply use the proposed return value. In other words, the math function will pass that value back to its caller.

The Lattice compiler package includes source code for **matherr** so you can change it to do more sophisticated error correction. One typical change is to place a different return value into the exception vector and then return a non-zero action code. This informs the math function that the return value has been changed.

The **except** function is a Lattice extension to UNIX that simplifies the interface to **matherr** by setting up the exception vector and processing the action code and return value. It is intended to ease the error-handling chore in user-written math functions.

When your math function encounters an error, it should call **except** specifying one of the following error types, which are defined in the **math.h** header file:

SYMBOL	CODE	MEANING
DOMAIN	1	Domain error
SING	2	Singularity
OVERFLOW	3	Overflow (number too large)
UNDERFLOW	4	Underflow (number too small)
TLOSS	5	Total loss of significance
PLOSS	6	Partial loss of significance

You can define new type codes if your application requires them, but you should then change **matherr** to perform the appropriate mapping into the UNIX error codes. The default mapping is:

matherr	errno
DOMAIN	EDOM
SING	EDOM
OVERFLOW	ERANGE
UNDERFLOW	ERANGE
TLOSS	ERANGE
PLOSS	ERANGE

RETURNS

For **matherr,** a non-zero return indicates that the proposed return value in the exception vector has been changed and that the new value should be used. A zero return indicates that the proposed return value is OK.

For **except,** the actual return value (a double) is passed back.

SEE ALSO

_fperr

Memory block operations **Class: UNIX**

NAME

SYNOPSIS

```
#include <string.h>

s = memccpy(to,from,c,n);
s = memchr(a,c,n);
x = memcmp(a,b,n);
s = memcpy(to,from,n);
s = memset(to,c,n);

movmem(from,to,n);
repmem(to,vt,nv,nt);
setmem(to,n,c);
swmem(a,b,n);

char *to;          destination pointer
char *from;        source pointer
unsigned n;        number of bytes
char c;            character value
char *a,*b;        block pointers
char *vt;          value template
int nv;            number of bytes in template
int nt;            number of templates in block
char *s;           return pointer
int x;             return value
```

Memory block operations **Class: UNIX**

DESCRIPTION

These functions manipulate blocks of memory in various ways.

The **memcpy** and **movmem** functions are similar, except the former was introduced with UNIX V, while the latter is a traditional Lattice function. In a like manner, **memset** and **setmem** perform the same operation, except that the former is UNIX-compatible. Note that **memcpy** and **memset** return a pointer to the destination block, while **movmem** and **setmem** have void returns. Also note that **memcpy** and **movmem** are smart enough to handle overlapping memory blocks correctly.

The **memccpy** function is similar to **memcpy** except that copying stops after the specified block size has been copied or after the specified character has been copied. It returns a pointer to the character after **c** in the **from** block, or a null pointer if **c** was not found in the first **n** characters. Unlike **memcpy**, **memccpy** does not handle overlapping memory blocks. If you specify overlapping blocks to this function, the results are unpredictable.

The **memchr** function returns a pointer to the first occurrence of the specified character in the block, or a null pointer if the character is not found.

The **memcmp** function performs a character-by-character comparison of two memory blocks and returns an integral value as follows:

```
NEGATIVE => first block is below second
ZERO     => first block equals second
POSITIVE => first block is above second
```

There is currently no UNIX equivalent for **swmem** and **repmem.** The former merely swaps two blocks in memory, although it has a major performance advantage over the typical for-loop approach. The latter replicates a template of values throughout a block and is very useful when you need to initialize an array of structures to some non-zero pattern.

RETURNS

As noted above

CAUTIONS

Remember that these functions neither recognize nor produce the null terminator byte usually found at the end of strings. A common mistake is to assume that **memcpy,** just like **strcpy,** automatically places a null byte at the end of the block. It doesn't.

Make a new directory **Class: UNIX**

NAME

mkdir...............Make a new directory

SYNOPSIS

```
#include <stdio.h>

error = mkdir(path);

int error;        0 if successful
char *path;       points to new directory path string
```

DESCRIPTION

This function makes a new directory in the specified path. For example, if **path** is "sys:/abc/def/ghi", then the new directory is named "ghi" and is in the path "/abc/def" on the system drive. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

onbreak

Plant break trap **Class: AMIGA**

NAME

onbreak............Plant break trap

SYNOPSIS

```
#include <dos.h>

error = onbreak(func);

int error;          0 if successful
int (*func)();      pointer to function to be called
```

DESCRIPTION

This function plants a break trap, which is simply a function that gets called whenever the user keys CTRL-C or CTRL-D. The function can perform any AmigaDOS operations. If it returns a value of Ø, then execution resumes at the interrupted point. Otherwise, the program is aborted immediately.

If **func** is null, then the current break trap, if any, is removed and the default interrupt handler is restored. With the default handler, CTRL-C and CTRL-D cause a requester to appear on the screen with choices of **continue** or **abort**.

Detection of CTRL-C and CTRL-D is performed during level 1 I/O. Explicit checks for these events can be forced by calls to the function **chkabort**.

RETURNS

The **onbreak** function returns Ø if it was successful. The break trap function should return Ø to continue execution and non-zero to abort.

Plant break trap **Class: AMIGA**

SEE ALSO

chkabort

EXAMPLES

```
/*
*
* This program tests the onbreak function.  After
* the initial message is printed, you should get
* the "Break received" message when you hit CTRL C
* or CTRL D. The second time will cause the program
* to terminate.
*
*/
#include <stdio.h>
int i = 0;

int brk()                /* This is the break function */
{
printf("Break received...\n");
return(i++);
}

main()                   /* This is the main program */
{
printf("Setting break trap...\n");
if(onbreak(&brk)) printf("Can't set break trap\n");
while(1) chkabort();     /* check for CTRL-C     */
}
```

onexit

Exit trap **Class: ANSI**

NAME

onexit..............Exit trap

SYNOPSIS

```
#include <stdlib.h>

success = onexit(func);

int success;     non-zero if successful
int (*func)();   pointer to trap function
```

DESCRIPTION

This function establishes a "trap" that will be called when the program terminates. The trap function is called just before the program returns to the operating system. For normal termination via the **exit** function or via a return from the **main** function, all buffers are flushed and files are closed before the trap is called. If the program is using **_exit,** the files and buffers may still be open, depending on what the program does before terminating. In both cases, user-allocated memory is not yet freed.

As an extension to the ANSI definition of this function, we pass the exit code to the trap function as its only argument. Then whatever value the trap function returns is used as the real exit code.

Another difference between our implementation and the ANSI definition is that we allow only one trap. Each call to **onexit** overrides the previous trap. If you call **onexit** with a null pointer, the current trap is removed.

Exit trap **Class: ANSI**

CAUTIONS

Remember that the exit trap is called after all files have been closed. A common mistake is to issue some type of output message via **printf** or **cprintf** from within the exit trap. In order for this to work, you should **fopen** or **open** the console device and send the message via **fprintf** or **write**.

SEE ALSO

exit, _exit

EXAMPLES

```
/*
*
* This program tests the "onexit" function.
*
*/
#include <stdio.h>

int ex(i)          /* This is the exit trap function */
int i;
{
FILE *con;

if((con = fopen("*","w")) != NULL)
  {
  fprintf(con,"Exit trap hit...code %d found\n",i);
  fclose(con);
  }
return(0);
}

main()             /* This tests the exit trap */
{
int (*p)();
```

```
    p = &ex;
    printf("Setting exit trap...\n");
    if(!onexit(p))  printf("Can't set trap...\n");
    printf("Exiting with code 2\n");
    exit(2);
    }
```

Open a Level 1 file **Class: UNIX**

NAME

open...............Open a Level 1 file

SYNOPSIS

```
#include <fcntl.h>

fh = open(name,mode,prot);

int fh;          file handle
char *name;      file name
int mode;        access mode
int prot;        protection mode (O_CREAT only)
```

DESCRIPTION

This function opens a file so that it can be accessed via the level 1 I/O functions. The file name can be any character string that is a valid file name, and it may include a device code and a directory path. The access mode is formed by ORing together the appropriate symbols from the following list:

CONVENTIONAL UNIX SYMBOLS...

O_RDONLY Read-only access. No writes are allowed.

O_WRONLY Write-only access. No reads are allowed.

O_RDWR Read-write access. Both reads and writes are allowed.

O_NDELAY This symbol is defined for UNIX compatibility and has no effect under AmigaDOS.

Open a Level 1 file **Class: UNIX**

O_APPEND	This symbol is normally used in conjunction with O_WRONLY or O_RDWR. It causes the I/O system to seek to the end of the file before each write operation. After each write operation, the file is positioned at the new end-of-file.
O_TRUNC	If the file exists, it is truncated to a length of Ø. This flag is normally used with O_CREAT, O_WRONLY or O_RDWR.
O_CREAT	If the file does not already exist, it is created. The protection mode argument is provided for compatibility with existing software, but is ignored under AmigaDOS. To set the protection use the **chmod** function to change the protection bits after the file has been closed.
O_EXCL	This symbol is used only with O_CREAT. If O_EXCL and O_CREAT are both present and the file already exists, the **open** function will fail.

RETURNS

If the operation is successful, the function returns a file handle, which is an integer equal to or greater than Ø. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, chgfa, chmod, close, creat

Print UNIX error message **Class: ANSI**

NAME

perror..............Print UNIX error message

SYNOPSIS

```
#include <stdio.h>

error = perror(s);

int error;        contents of errno
char *s;          message prefix
```

DESCRIPTION

This function checks **errno** and, if it is non-zero, sends an error message to **stderr.** The message consists of the specified prefix, a colon and space, and the message text from the external array named **sys_errlist.** This array contains pointers to the various UNIX error messages. The highest error number is given by the contents of external integer **sys_nerr.** The Lattice compiler package contains the source for these two external items in a file named **syserr** so you can change or expand the messages as you desire.

RETURNS

The function returns the contents of **errno** so you can test for an error condition and print a message in one step, as in the example:

```
if(perror("foo")) goto abort;
```

Note that this feature is an extension of UNIX, and if you use it, your program may be non-portable.

Print UNIX error message **Class: ANSI**

SEE ALSO

errno, sys_nerr, sys_errlist, poserr

Print AmigaDOS error message **Class: AMIGA**

NAME

poserr..............Print AmigaDOS error message

SYNOPSIS

```
#include <dos.h>

error = poserr(s);

int error;          contents of _OSERR
char *s;            message prefix
```

DESCRIPTION

This function checks **_OSERR** and, if it is non-zero, sends an error message to **stderr**. The message consists of the specified prefix, a colon and space, and the message text from the external array named **os_errlist**. This array of structures contains pointers to the various AmigaDOS error messages. The highest error number is given by the contents of external integer **os_nerr**. The Lattice compiler package contains the source for these two external items in a file named **oserr.c** so you can change or expand the messages.

RETURNS

The function returns the contents of **_OSERR** so you can test for an error condition and print a message in one step, as in the example:

```
    if(poserr("foo")) goto abort;
```

SEE ALSO

_OSERR, os_errlist, os_nerr, perror

qsort

Sort a data array **Class: UNIX**

NAME

```
qsort..............Sort a data array
dqsort.............Sort an array of doubles
fqsort.............Sort an array of floats
lqsort.............Sort an array of long integers
sqsort.............Sort an array of short integers
tqsort.............Sort an array of text pointers
```

SYNOPSIS

```
#include <stdlib.h>

qsort(a,n,size,cmp);
dqsort(da,n);
fqsort(fa,n);
lqsort(la,n);
sqsort(sa,n);
tqsort(ta,n);

char *a;          data array pointer
double *da;       pointer to double array
float *fa;        pointer to float array
long *la;         pointer to long int array
short *sa;        pointer to short int array
char *ta[];       pointer to text pointer array

int n;            number of elements in array
int size;         element size in bytes
int (*cmp)();     pointer to comparison function
```

DESCRIPTION

The **qsort** function sorts the specified data array using the ACM 271 algorithm popularly known as Quick-sort. During its operation, it calls upon the specified comparison routine with pointers to the two array elements being compared. As an extension to the

normal UNIX implementation, we also pass a third argument, the element size, which can be ignored. The comparison routine should return an integral result as follows:

```
NEGATIVE => first element is below second
POSITIVE => first element is above second
ZERO     => elements are equal
```

The **dqsort, fqsort, lqsort, sqsort,** and **tqsort** functions sort various arrays that are commonly encountered. They are all straightforward except for **tqsort,** which needs some explanation. The **ta** array consists of pointers to null-terminated character strings. The **tqsort** function re-arranges the pointers so that the strings are in ascending ASCII sequence, using **strcmp** as the comparison routine. Note that it is not the pointer values that are being sorted, but the strings to which the pointers refer.

rand,srand

Simple random number generator **Class: ANSI**

NAME

rand................Generate a random number
srand...............Set seed for rand function

SYNOPSIS

```
#include <stdlib.h>

x = rand();
srand(seed);

int x;          random number
unsigned seed;  random number seed
```

DESCRIPTION

The **rand** function returns pseudo-random numbers in the range from 0 to the maximum positive integer value. When you call **srand,** the random number generator is reset to a new seed value. The initial default seed is 1.

See **drand48** and its related functions for more sophisticated random number generation.

RETURNS

As noted above.

SEE ALSO

drand48

EXAMPLES

```
/*
*
```

Simple random number generator **Class: ANSI**

```
* This example prints 1000 random numbers
*
*/
#include <stdio.h>
#include <stdlib.h>

main(argc,argv)
int argc;
char *argv[];
{
int i;
unsigned x;

if(argc > 1)
  {
  stcd_u(argv[1],&x);
  if(x == 0) x = 1;
  printf("Seed value is %d\n",x);
  srand(x);
  }
printf("Here are 1000 random numbers...\n");
for(i = 0; i < 200; i++)
  printf("%5d %5d %5d %5d %5d\n",
         rand(),rand(),rand(),rand(),rand());
printf("\n\n");
}
```

Read or write Level 1 file **Class: UNIX**

NAME

read................Read from Level 1 file
write...............Write to Level 1 file

SYNOPSIS

```
#include <fcntl.h>

count = read(fh,buffer,length);
count = write(fh,buffer,length);

int count;        actual bytes read or written
int fh;           file handle
char *buffer;     data buffer
int length;       number of bytes to read or write
```

DESCRIPTION

These functions read or write a Level 1 file whose handle was returned by **creat** or **open.** Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult **errno** and **_OSERR.** If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check **errno** and **_OSERR** just in case some malfunction caused the short count.

Note that these functions are very similar to the Level 0 I/O functions **dread** and **dwrite.** The primary difference is that Level 1 files will be automatically closed by **exit** and **_exit,** which are usually called for

you when the program terminates.

RETURNS

If the operation is successful, the function returns the actual number of bytes transferred. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, open, dread, dwrite

remove,unlink

Remove a file **Class: ANSI**

NAME

remove..............Remove a file
unlink..............Remove a file

SYNOPSIS

```
#include <stdio.h>

error = remove(name);
error = unlink(name);

int error;        non-zero if error
char *name;       file name
```

DESCRIPTION

These functions remove the specified file from the system. They behave identically, but **unlink** is provided for compatibility with some versions of UNIX. The **remove** function is preferred because it is now in the ANSI C standard.

The file name argument can include a path, but it cannot include wild card characters. That is, you can remove only one file at a time.

RETURNS

If a non-zero value is returned, some type of error occurred, and additional information can be found in **errno** and **_OSERR.** The most common errors occur when you try to remove a file that doesn't exist or that is marked as read-only.

SEE ALSO

errno, _OSERR

Remove a file Class: ANSI

EXAMPLES

```
/*
*
* This program removes all files specified in the
* argument list.  It does not allow wild card
* characters in the file names.
*
*/
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int i;              /* loop counter */
int ret = 0;        /* exit code, non-zero if any failures */

for(i = 1; i < argc; i++) if(remove(argv[i])
  {
  perror("RMV");
  ret = 1;
  }
exit(ret);
}
```

Rename a file **Class: ANSI**

NAME

rename..............Rename a file

SYNOPSIS

```
#include <stdio.h>

error = rename(old,new);

int error;        0 for success, -1 for error
char *old;        old file name
char *new;        new file name
```

DESCRIPTION

This function renames a file, if possible. The old name can include a path, but the new name should not. A failure occurs if the old file cannot be found or if the new name is the same as an existing file.

RETURNS

If the function fails, it returns -1 and places additional error information into **errno** and **_OSERR**. Success is indicated by a return value of Ø.

EXAMPLES

```
/*
*
* This is a version of the "rename" command
* that prompts for the old and new names.
*
*/
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
```

Rename a file **Class: ANSI**

```
main(argc,argv)
int argc;
char *argv[];
{
char old[FMSIZE],new[FMSIZE];
char *pold,*pnew;

if(argc < 2)               /* Get old file name */
  {
  printf("OLD FILE: ");
  if(gets(old) == NULL) exit(1);
  pold = old;
  }
else pold = argv[1];

if(arc < 3)
  {
  printf("NEW FILE: ");
  if(gets(new) == NULL) exit(1);
  pnew = new;
  }
else pnew = argv[2];

if(rename(pold,pnew))
  {
  perror("RENAME");
  exit(1);
  }
}
```

rlsmem,rlsml

Release a Level 2 memory block **Class: LATTICE**

NAME

SYNOPSIS

```
#include <stdlib.h>

error = rlsmem(p,sbytes);
error = rlsml(p,lbytes);

int error;         non-zero if error
char *p;           block pointer
short sbytes;      number of bytes
long lbytes;       number of bytes
```

DESCRIPTION

These functions release memory blocks that were previously obtained via **getmem** or **getml**. If the number of bytes is less than the value **MELTSIZE** as defined in header file **dos.h**, then it is made equal to **MELTSIZE**.

Note that you can free a portion of a block as long as the portion being freed is at least **MELTSIZE** bytes long. For example, suppose you allocate a block of 100 bytes, use the lower 80, and decide to return the upper 20 to the memory pool. Simply make the following call:

```
rlsmem(p+80,20);
```

where **p** is the original block pointer. Again, whenever you free partial blocks in this way, make sure that the portion being freed is equal to or greater than **MELTSIZE**.

Release a Level 2 memory block **Class: LATTICE**

RETURNS

If the block is not in the current Level 2 memory pool or overlaps a block that is already free, a value of -1 is returned. Otherwise, the return value is Ø.

EXAMPLES

```
/*
*
* This program builds a linked list of text strings
* obtained from the standard input file.
*
*/
#include <stdio.h>
#include <stdlib.h>
/*
*
* These elements are linked together to form the
* text string list.
*
*/
#define MAX 256
struct LIST
  {
  struct LIST *next;     /* forward linkage */
  int size;              /* element size */
  char text[MAX];        /* minimum text string */
  };
/*
*
* Main program
*
*/
main()
{
struct LIST *p,*q,*list = NULL;
char b[MAX];
```

```
int x;
/*
*
* Build the list
*
*/
for(q = (struct LIST *)(&list);;q = p)
  {
  printf("Enter a text string: ");
  if(gets(b) == NULL) break;
  if(b[0] == '\0') break;
  p = (struct LIST *)getmem(sizeof(struct LIST));
  if(p == NULL)
    {
    printf("No more memory\n");
    break;
    }
  p->next = q->next;
  x = MAX - (stccpy(p->text,b,256);
  p->size = sizeof(struct LIST) - x;
  if(x >= MELTSIZE) rlsmem(p+p->size,x);
  }
/*
*
* Print the list
*
*/
printf("\n\nTEXT LIST...\n");
for(p = list; p != NULL; p = p->next)
   printf("%s",p->text);
}
```

Remove a directory **Class: UNIX**

NAME

rmdir...............Remove a directory

SYNOPSIS

```
#include <stdio.h>

error = rmdir(path);

int error;        0 if successful
char *path;       points to directory path string
```

DESCRIPTION

This function removes an existing directory in the specified path. For example, if **path** is "sys:/abc/def/ghi", then the directory named "ghi" is removed from the path "/abc/def" on the system drive. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

Set buffer mode for L2 file **Class: ANSI**

NAME

setbuf..............Set buffer mode for L2 file
setnbf..............Set non-buffer mode for L2 file
setvbuf.............Set variable buffer for L2 file

SYNOPSIS

```
#include <stdio.h>

setbuf(fp,buff);
setnbf(fp);
error = setvbuf(fp,buff,type,size);

int error;        0 if successful
FILE *fp;         file pointer
char *buff;       buffer pointer
int type;         type of buffering
int size;         buffer size in bytes
```

DESCRIPTION

These functions set the buffering mode for a Level 2 file. Proper usage is to call the appropriate function after calling **fopen** and before calling any other Level 2 I/O functions. If you fail to follow this rule, the file may become corrupted.

The Level 2 I/O system automatically allocates a buffer via **getmem** when you perform the first read or write operation. Then the data being read or written is staged through this buffer in order to improve I/O efficiency. If you would rather use your own buffer instead of having one allocated for you, call **setbuf** with a non-null buffer pointer. The buffer size must be at least as large as the value given in the external integer **_bufsiz**, which defaults to the value of the symbol **BUFSIZ**, defined in **stdio.h**.

Set buffer mode for L2 file **Class: ANSI**

You can eliminate buffered I/O by calling **setnbf** or by calling **setbuf** with a null buffer pointer. When this is done, physical I/O occurs whenever your program performs Level 2 read or write operation, even if only one byte is being transferred. This is very inefficient for disk files but often desirable for terminal or communication ports.

The **setvbuf** function can do everything that the other two functions can do, and it can also set "line buffered" mode and attach a buffer of non-standard size. The **type** argument must be one of the following symbols defined in **stdio.h:**

```
_IOFBF => Fully buffered
_IOLBF => Line buffered
_IONBF => Non-buffered
```

For **_IOFBF** and **_IOLBF,** the specified buffer will be attached to the file unless **buff** is null, in which case a buffer will be automatically allocated on the first read or write. For the **_IONBF** case, the **buff** and **size** arguments are ignored.

The line-buffered mode is useful for interactive applications. When in this mode, the buffer is flushed whenever a newline is sent, the buffer is full, or input is requested. Note, however, that you must use the **fputc** and **fputchar** functions instead of the **putc** and **putchar** macros in order for line buffering to work correctly. The macros do not check if line-buffered mode is active, and so they behave as if the file were fully buffered.

RETURNS

For **setvbuf,** the error code is non-zero if **type** or

size is invalid.

CAUTIONS

These functions must be used only after **fopen** and before any other Level 2 file operations. Also, a common error is to allocate a buffer on the stack within a function, attach it to a file, and then return from the function. This will corrupt the stack.

SEE ALSO

fopen

Establish event traps **Class: ANSI**

NAME

signal..............Establish event traps

SYNOPSIS

```
#include <signal.h>

oldfun = signal(sig,newfun);

int (*oldfun)();          old trap function
int sig;                  signal number
int (*newfun)();          new trap function
```

DESCRIPTION

This function establish traps for various events that can occur outside of your program.

The **newfun** argument specifies the action to be taken when the signal occurs, as follows:

SIG_IGN Ignore the signal.

SIG_DFL Take the system default action, as indicated above for each signal.

If **newfun** is not any of the above, then it must be a valid function pointer. When the signal is detected, the action is reset to either SIG_DFL or SIG_IGN, depending on the particular signal. Then the trap function is called with an integer argument specifying which signal was detected (e.g. SIGINT). The trap function can take whatever action is necessary, including calling **signal** again to re-establish itself as the trap function. If the function returns, execution continues at the point in your program where the signal was detected.

Establish event traps **Class: ANSI**

The **sig** argument specifies which signal is being trapped, using the following symbols defined in **signal.h:**

SIGFPE This signal occurs whenever a floating point error is detected and the standard version of **CXFERR** is installed. If you install your own version, you must duplicate our code (supplied as a file named CXFERR.C) in order to provide this signal.

SIGINT This signal occurs whenever the user operates the CTRL-C or CTRL-D key combination. The default action for AmigaDOS is to abort your program. If you specify a function to be called, the signal will be reset to SIG_IGN when the interrupt occurs. Your function should call **signal** again if you want to reinstall the trap.

RETURNS

The **signal** function normally returns the previous value of the trap function, which may be SIG_IGN or SIG_DFL.

Get Level 2 memory pool size **Class: LATTICE**

NAME

sizmem..............Get Level 2 memory pool size

SYNOPSIS

```
#include <stdlib.h>

size = sizmem();

long size;
```

DESCRIPTION

This function returns the number of unallocated bytes in the current Level 2 memory pool. This value is the sum of the sizes of all unallocated blocks, and so it does not indicate the size of the largest free block.

Also, the value does not indicate the maximum amount of Level 2 memory that can be allocated. That is, the Level 2 allocation functions **getmem** and **getml** will automatically expand the pool via **sbrk** when no block of sufficient size is found in the pool.

SEE ALSO

getmem, getml, rlsmem, rlsml, rstmem

Get an argument **Class: LATTICE**

NAME

stcarg..............Get an argument

SYNOPSIS

```
#include <string.h>

length = stcarg(s,b);

int length;       number of bytes in argument
char *s;          text string pointer
char *b;          break string pointer
```

DESCRIPTION

This function scans the text string until one of the break characters is found or until the null terminating byte is hit. While scanning, **stcarg** skips over substrings that are enclosed in single or double quotes, and the backslash is recognized as an escape character. In other words, break characters will not be detected if they are quoted or preceded by a backslash.

RETURNS

The function returns a count of the number of characters in **s** up to but not including the break character or null terminator.

SEE ALSO

stpbrk,strcspn,strpbrk

EXAMPLES

```
#include <stdio.h>
```

Get an argument **Class: LATTICE**

```
#include <string.h>

main()
{
char a[256],b[256];
int x;

while(1)
  {
  printf("Enter text string:  ");
  if(gets(a) == NULL) exit(0);
  printf("Enter break string: ");
  if(gets(b) == NULL) exit(0);
  x = stcarg(a,b);
  printf("Arg length: %d, Arg text: %.*s\n",x,x,a);
  }
}
```

Copy one string to another **Class: ANSI**

NAME

```
stccpy.............Copy one string to another
stpcpy.............Copy one string to another
strcpy.............Copy one string to another
strncpy............Copy string, length-limited
```

SYNOPSIS

```
#include <string.h>

size = stccpy(to,from,n)
np = stpcpy(to,from);
p = strcpy(to,from);
p = strncpy(to,from,n);

char *np;        points to end of destination string
char *p;         same as destination pointer
char *to;        destination pointer
char *from;      source pointer
int n;           maximum source length
```

DESCRIPTION

These functions copy the null-terminated source string to the destination area. For **stpcpy** and **strcpy,** the entire source string is copied, and the resulting destination is always null-terminated. The **strncpy** function always writes exactly **n** characters to the destination. If the null terminator is hit before **n** characters are copied from the source, then the destination is filled with null bytes. If the source string contains more than **n** non-null characters, the destination will not be null-terminated.

The **stccpy** function is similar to **strncpy** except that it always produces a null-terminated string, and it returns actual number of bytes placed in the **to** area,

including the null terminator.

RETURNS

The **strcpy** and **strncpy** functions return a pointer that is the same as the destination pointer. The Lattice function **stpcpy** returns a pointer to the end of the destination string, which is often more useful when you are building a string up from several pieces.

CAUTIONS

Be careful when using **strncpy,** since it is one of the few string functions that do not always produce a null-terminated string.

EXAMPLES

```
/*
*
* This example should print: Hello, my name is Flo.
*
*/
#include <string.h>

main()
{
char b[256],*p;

p = stpcpy(b,"Hello, ");
p = stpcpy(p,"my name ");
p = stpcpy(p,"Flo.");
puts(b);
}
```

Convert strings to integer **Class: LATTICE**

NAME

stcd_i..............Convert decimal string to int
stco_i..............Convert octal string to int
stch_i..............Convert hexadecimal string to int
stcd_l..............Convert decimal string to long int
stco_l..............Convert octal string to long int
stch_l..............Convert hexadecimal string to long

SYNOPSIS

```
#include <string.h>

length = stcd_i(in,ivalue);
length = stci_o(in,ivalue);
length = stci_h(in,ivalue);
length = stcl_d(in,lvalue);
length = stcl_o(in,lvalue);
length = stcl_h(in,lvalue);

int length;              input length
char *in;                input string pointer
int *ivalue;             integer value pointer
long *lvalue;            long integer value pointer
```

DESCRIPTION

These functions scan an input string and convert the leading characters into short or long integers. For **stcd_i** and **stcd_l**, the input string must begin with a plus sign '+', minus sign '-', or a decimal digit ('Ø' to '9'). The octal conversions **stco_i** and **stco_l** process an unsigned string of octal digits ('Ø' to '7'). Finally, the hexadecimal conversions **stch_i** and ,bd stch_l handle unsigned strings containing digits from 'Ø' to '9' and letters from 'A' to 'F' or 'a' to 'f'. Scanning of the input string stops when the first invalid character is reached, At that point, the

resulting value is stored into the area addressed by the second argument.

RETURNS

Each function returns the number of input characters converted. This result will be Ø if the first character of the input string is not valid for the particular conversion. In that case, conversion result stored via the second argument will be Ø.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
int x;
long j;
char b[80];

while(1)
   {
   printf("\nEnter a hexadecimal value: ");
   if(gets(b) == NULL) exit(0);
   x = stch_l(b,&j);
   printf("stch_l: Length %d, Result %lx\n",x,j);
   }
}
```

stcgfe,stcgfn,stcgfp

Get file name components **Class: LATTICE**

NAME

stcgfe..............Get file extension
stcgfn..............Get file node
stcgfp..............Get file path

SYNOPSIS

```
#include <string.h>

size = stcgfe(ext,name);
size = stcgfn(node,name);
size = stcgfp(path,name);

int size;         size of result string
char *ext;        extension area pointer
char *node;       node area pointer
char *path;       path area pointer
char *name;       file name pointer
```

DESCRIPTION

These functions isolate the path, node, or extension portion of a file name. The node is the rightmost portion of the file name that is separated from the rest of the name by a colon, slash, or backslash. The extension is the final part of the node that begins with a period, and the path is the leading part of the name up to the node. For example,

NAME	PATH	NODE	EXTENSION
"myprog.c"	""	"myprog.c"	"c"
"/abc.dir/def"	"/abc.dir"	"def"	""
"/abc.dir/def.ghi"	"/abc.dir"	"def.ghi"	"ghi"
"dfØ:yourfile"	"dfØ:"	"yourfile"	""
"/abc/"	"/abc"	""	""

RETURNS

The **size** value is the same as would be returned by the **strlen** function. That is, if **size** is Ø, then the desired portion of the file name could not be found and the result area contains a null string.

SEE ALSO

strsfn

Convert integers to strings **Class: LATTICE**

NAME

```
stci_d..............Convert int to decimal
stci_o..............Convert int to octal
stci_h..............Convert int to hexadecimal
stcl_d..............Convert long int to decimal
stcl_o..............Convert long int to octal
stcl_h..............Convert long int to hexadecimal
stcu_d..............Convert unsigned int to decimal
stcul_d.............Convert unsigned long to decimal
```

SYNOPSIS

```
#include <string.h>

length = stci_d(out,ivalue);
length = stci_o(out,ivalue);
length = stci_h(out,ivalue);
length = stcl_d(out,lvalue);
length = stcl_o(out,lvalue);
length = stcl_h(out,lvalue);
length = stcu_d(out,uivalue);
length = stcul_d(out,ulvalue);

int length;                 output length
char *out;                  output buffer pointer
int ivalue;                 integer value
long lvalue;                long integer value
unsigned int uivalue;       unsigned integer value
unsigned long ulvalue;      unsigned long integer value
```

DESCRIPTION

These functions convert various integral values into ASCII strings. The output area must be large enough to accomodate the maximum possible string, including the terminating null byte that each function appends. The following table shows the required lengths.

Convert integers to strings **Class: LATTICE**

FUNCTION	LENGTH
stci_d	7
stci_o	7
stci_h	5
stcl_d	13
stcl_o	12
stcl_h	9
stcu_d	6
stcul_d	12

For **stci_d** and **stcl_d**, the first output character will be a minus sign if the input value is negative. No special leading character is generated if the value is positive. For all functions, leading zeroes are suppressed, and a single 'Ø' character is generated if the input value is Ø.

RETURNS

The return value is the number of characters actually placed into the output area, not including the final null byte.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
int i,x;
char b[13];

while(1)
        {
        printf("\nEnter a short integer: ");
```

Convert integers to strings **Class: LATTICE**

```
        scanf("%d",&i);
        x = stci_d(b,i);
        printf("stci_d: Length %d, Result %s\n",x,b);
        x = stci_o(b,i);
        printf("stci_o: Length %d, Result %s\n",x,b);
        x = stci_h(b,i);
        printf("stci_h: Length %d, Result %s\n",x,b);
        }
}
```

Measure character set span **Class: ANSI**

NAME

stcis...............Measure span of chars in set
stcisn..............Measure span of chars not in set
strspn..............Measure span of chars in set
strcspn.............Measure span of chars not in set

SYNOPSIS

```
#include <string.h>

length = stcis(s,b);
length = stcisn(s,b);
length = strspn(s,b);
length = strcspn(s,b);

int length;     span length in bytes
char *s;        points to string being scanned
char *b;        points to character set string
```

DESCRIPTION

These functions measure the number of characters at the beginning of input string **s** that are either in or not in the character set specified by **b**. The **stcis** and **strspn** functions are identical and count the number of leading characters that are in the set. Likewise, **stcisn** and **strcspn** are identical and count the number of leading characters that are not in the set. The "stc" pair are provided for compatibility with other version of Lattice C, while the "str" functions are now part of the ANSI standard.

RETURNS

The functions all return the number of bytes that are in or not in the specified character set. Note that the scan always stops when the null terminator byte is

reached.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char s1[256],s2[256];

while(1)
        {
        printf("\nEnter test string: ");
        if(gets(s1) == NULL) exit(0);
        printf("Enter span string: ");
        if(gets(s2) == NULL) exit(0);
        printf("strspn: %d\n",strspn(s1,s2));
        printf("strcspn: %d\n",strcspn(s1,s2));
        printf("stcis: %d\n",stcis(s1,s2));
        printf("stcisn: %d\n",stcisn(s1,s2));
        }
}
```

NAME

stcpm..............Unanchored pattern match
stcpma.............Anchored patter match

SYNOPSIS

```
#include <string.h>

size = stcpm(string,pattern,match);
size = stcpma(string,pattern);

int size;        size of matching string
char *string;    string to be scanned
char *pattern;   pattern string
char **match;    returns pointer to matching string
```

DESCRIPTION

These functions scan a string to find a specified pattern. The pattern is specified in a simplified form of regular expression notation where

?	matches any single character
c*	matches zero or more occurrences of character c
c+	matches one or more occurrences of character c
\?	matches a question mark (?)
*	matches an asterisk (*)
\+	matches a plus sign (+)

Any other character must match exactly. Some examples are

Pattern match functions **Class: LATTICE**

PATTERN	**MATCHES**
"abc"	Only "abc"
"ab*c"	"ac" or "abc" or "abbc" and so on
"ab+c"	"abc" or "abbc" or "abbbc" and so on
"ab?*c"	Any string starting with "ab" and ending with "c" such as "abxyzc"
"ab*c"	Only "ab*c"

For **stcpma,** the match must occur at the beginning of the string, while for **stcpm,** the match can occur anywhere in the string. In either case, the function returns the size of the matching string or zero if there was no match. Also, **stcpm** returns a pointer to the beginning of the matching string.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char s[100],p[100],*r;
int x;

while(1)
  {
  printf("\nSearch string => ");
  if(gets(s) == NULL) break;
  printf("Pattern       => ");
  if(gets(p) == NULL) break;
  x = stcpma(s,p);
  if(x)
    printf("stcpma: size = %d, match = \"%.*s\"\n",x,x,s);
  else
    printf("stcpma: no match\n"):
```

Pattern match functions **Class: LATTICE**

```
    x = stcpm(s,p,&r);
    if(x)
      printf("stcpm: size = %d, match = \"%.*s\"\n",x,x,r);
    else
      printf("stcpm: no match\n");
    }
  }
```

Skip blanks (white space) **Class: LATTICE**

NAME

stpblk..............Skip blanks (white space)

SYNOPSIS

```
#include <string.h>

q = stpblk(p);

char *q;          updated string pointer
char *p;          string pointer
```

DESCRIPTION

This function advances the string pointer past white space characters, that is, past all the characters for which **isspace** is true.

RETURNS

The function returns a pointer to the next non-white-space character. Note that the null terminator byte is not considered to be white space, and so the function will not go past the end of the string.

SEE ALSO

stcis, strspn

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char input[256];
```

Skip blanks (white space) **Class: LATTICE**

```
    while(1)
        {
        puts("\nEnter a string with leading blanks...");
        if(gets(input) == NULL) exit(0);
        printf("%s\n",stpblk(input));
        }
    }
```

stpbrk,strpbrk

Find break character in string **Class: ANSI**

NAME

stpbrk..............Find break character in string
strpbrk.............Find break character in string

SYNOPSIS

```
#include <string.h>

p = stpbrk(s,b);
p = strpbrk(s,b);

char *p;        points to break character in s
char *s;        string to be scanned
char *b;        break characters
```

DESCRIPTION

These functions scan string **s** to find the first occurrence of a character from break string **b**. They are completely equivalent, except that **strpbrk** is the UNIX name, while **stpbrk** is the traditional Lattice name.

RETURNS

If no character from **b** is found in **s**, a NULL pointer is returned. Otherwise, **p** is a pointer to the break first break character.

SEE ALSO

strspn, strcspn

EXAMPLES

```
#include <string.h>

/*
```

Find break character in string **Class: ANSI**

```
    *
    * Scan for commas, periods, and blanks.  Display
    * the tail of the string each time a break
    * character is found.
    *
    */
    char *p,s[] = "Hello, my name is Herkimer Q. Jerkimer."

    for(p = s; (p = strspn(p,",. ")) != NULL;
        printf("%s\n",p));
```

stpchr,stpchrn,strchr,strrchr

Find character in string **Class: ANSI**

NAME

stpchr..............Find character in string
stpchrn.............Find character not in string
strchr..............Find character in string
strrchr.............Find character not in string

SYNOPSIS

```
#include <string.h>

p = stpchr(s,c);
p = stpchrn(s,c);
p = strchr(s,c);
p = strrchr(s,c);

char *p;        updated string pointer
char *s;        input string pointer
char c;         character to be located
```

DESCRIPTION

The **stpchr** and **strchr** functions scan the input string to find the first occurrence of the character specified by argument **c**. Similarly, **stpchrn** and **strrchr** scan for the first occurrence of some character other than **c**. The "stp" versions are provided for compatibility with other versions of Lattice C, while the "str" functions are now part of the ANSI standard.

RETURNS

For **stpchr** and **strchr**, a NULL pointer is returned if the input string is empty or if the specified character is not found. The other two functions return a NULL pointer if the input string is empty or consists entirely of character **c**.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char c,s[256];

while(1)
        {
        printf("\nEnter test string: ");
        if(gets(s1) == NULL) exit(0);
        printf("Enter character: ");
        if((c = getchar()) == EOF) exit(0);
        printf("stpchr: %s\n",stpchr(s,c));
        printf("stpchrn: %s\n",strcspn(s,c));
        printf("strchr: %s\n",stcis(s,c));
        printf("strrchr: %s\n",stcisn(s,c));
        }
}
```

Convert date array to string **Class: LATTICE**

NAME

stpdate............Convert date array to string

SYNOPSIS

```
#include &lt;string.h&gt;

np = stpdate(p,mode,date);

char *np;         updated output string pointer
char *p;          output string pointer
int mode;         conversion mode
char *date;       date array, as follows
                    date[0] => year - 1980
                    date[1] => month (1 to 12)
                    date[2] => day (1 to 31)
```

DESCRIPTION

This function converts a 3-byte date array into ASCII or BCD according to the **mode** argument:

```
0 => yymmdd (BCD, 3 bytes)
1 => yymmdd (ASCII, 7 bytes)
2 => mm/dd/yy (ASCII, 9 bytes)
3 => mm-dd-yy (ASCII, 9 bytes)
4 => MMM d, yyyy (ASCII, up to 13 bytes)
5 => Mm...m d, yyyy (ASCII, up to 19 bytes)
6 => dd MMM yy (ASCII, 10 bytes)
7 => dd MMM yyyy (ASCII, 12 bytes)
```

In the above formats, MMM represents a 3-character month abbreviation in capitals, and Mm...m represents the full month name (e.g. January). The mm, dd, and yy terms are 2-character month, day, and year, respectively, while d is the date with the leading zero suppressed. The yyyy term is the 4-character year

obtained by adding 1980 to the first byte of the date array.

For all modes except 0, a null byte is appended to the output string.

RETURNS

The function does not make validity checks on the date array, and so it cannot fail. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the null terminator.

SEE ALSO

stptime, getclk, getft

Get next symbol from a string **Class: LATTICE**

NAME

stpsym..............Get next symbol from a string

SYNOPSIS

```
#include <string.h>

p = stpsym(s,sym,symlen);

char *p;        points to next input character
char *s;        input string
char *sym;      output string
int symlen;     sizeof(sym)
```

DESCRIPTION

This function breaks out the next symbol from the input string. The first character of the symbol must be alphabetic (upper or lower case), and the remaining characters must be alphanumeric. Note that the pointer is not advanced past any initial white space in the input string.

The output string is the null-terminated symbol, and will be an empty string if no symbol is found. If the symbol is longer than **symlen-1**, its excess characters are dropped.

RETURNS

The function returns a pointer to the next character past the symbol.

SEE ALSO

stcarg,stpbrk,strcspn,strpbrk

Get next symbol from a string **Class: LATTICE**

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char a[256],b[10];

while(1)
  {
  printf("\nEnter text string:  ");
  if(gets(a) == NULL) exit(0);
  While(1)
    {
    p = stpsym(a,b,sizeof(b));
    printf("Symbol: \"%s\"  Residual: \"%s\"\n",b,p);
    if(b[0] == '\0') break;
    }
  }
}
```

Convert time array to string **Class: LATTICE**

NAME

stptime............Convert time array to string

SYNOPSIS

```
#include <string.h>

np = stptime(p,mode,time);

char *np;        updated output string pointer
char *p;         output string pointer
int mode;        conversion mode
char *time;      time array, as follows
                   time[0] => hour (0 to 23)
                   time[1] => minute (0 to 59)
                   time[2] => second (0 to 59)
                   time[3] => hundredths (0 to 99)
```

DESCRIPTION

This function converts a 4-byte time array into ASCII or BCD according to the **mode** argument:

```
0 => hhmmssdd (BCD, 4 bytes)
1 => hhmmss (ASCII, 7 bytes)
2 => hh:mm:ss (ASCII, 9 bytes)
3 => hhmmssdd (ASCII, 9 bytes)
4 => hh:mm:ss.dd (ASCII, 12 bytes)
5 => hh:mm (ASCII, 6 bytes)
6 => hr:mm:ss HH (ASCII, 12 bytes)
7 => hr:mm HH (ASCII, 9 bytes)
```

The hh, mm, ss, and dd terms are simply the 2-digit (BCD or ASCII) equivalents of the binary values in the time array. The hr term is the 2-digit hour using the 12-hour form, and the HH term is either AM or PM.

Note that a null terminator is appended to the ASCII output strings.

RETURNS

The function does not make validity checks on the time array, and so it cannot fail. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the null terminator.

SEE ALSO

stpdate, getclk, getft

stptok

Get next token from a string **Class: LATTICE**

NAME

stptok..............Get next token from a string

SYNOPSIS

```
#include <string.h>

p = stptok(s,tok,toklen,brk);

char *p;        points to next character after token
char *s;        points to input string
char *tok;      points to output buffer
int toklen;     sizeof(tok)
char *brk;      break string
```

DESCRIPTION

This function breaks out the next token from the input string and moves it to the token buffer with a null terminator. A token consists of all characters in the input string **s** up to but not including the first character that is in the break string. In other words, **brk** specifies the characters that cannot be included in a token.

If the input string begins with a break character, then the token buffer will contain a null string, and the return pointer **p** will be the same as **s**. If no break character is found after **toklen-1** input characters have been moved to the token buffer, or if the input string terminator (a null byte) is hit, then the scan stops as if a break character were hit.

RETURNS

The function returns a pointer to the next character in the input string.

CAUTIONS

This function does not delete white space at the beginning of the input string.

SEE ALSO

stpblk, strtok

EXAMPLES

```
/*
*
* This example breaks out words that are
* separated by blanks or commas.
* The token buffer takes on the following
* values as the program loops:
*
*         LOOP    TOKEN
*          1      "first"
*          2      "second"
*          3      "third"
*          4      "fourth"
*
*/
#include <string.h>
#include <stdio.h>

char test[] = "first, second third, fourth";

main()
{
char *p = test;
char token[50];

while(1)
        {
        p = stptok(p,token,sizeof(token)," ,");
```

```
            printf("%s\n",token);
            if(*p == '\0') break;
            p = stpblk(++p);
            }
    }
```

NAME

strbpl..............Build string pointer list

SYNOPSIS

```
#include <string.h>

n = strbpl(s,max,t);

int n;           number of pointers
char *s[];       pointer to string pointer list
int max;         maximum number of pointers
char *t;         text pointer
```

DESCRIPTION

This function constructs a list of pointers to the strings contained within the specified text array. Each string must be null-terminated, and the text array must be terminated by a null string. In other words, array **t** must end with two null bytes, one to terminate the final string and another to terminate the array. The string pointer list **s** is terminated by a null pointer.

RETURNS

The return value indicates how many string pointers were placed into array **s**. If the number of strings plus the final null pointer is greater than **max**, a value of -1 is returned.

SEE ALSO

getfnl, strsrt

EXAMPLES

```
char text[] = {"string 1","string 2",'\0'};
char *list[5];
int count;

/*
*       The following call has the following effect:
*
*       Return value (count) is 2.
*       list[0] => "string 1"
*       list[1] => "string 2"
*       list[2] => NULL
*
*/
count = strbpl(list,5,text);
```

Concatenate strings **Class: ANSI**

NAME

strcat..............Concatenate strings
strncat.............Concatenate strings, max length

SYNOPSIS

```
#include <string.h>

p = strcat(to,from);
p = strncat(to,from,n);

char *p;         same as destination string pointer
char *to;        destination string pointer
char *from;      source string pointer
int n;           maximum source length
```

DESCRIPTION

These functions concatenate the source string to the tail end of the destination string. For **strncat,** no more than **n** characters are moved from the source to the destination. A null byte is placed at the end of the destination in any case.

RETURNS

Both functions return a pointer that is the same as the first argument.

SEE ALSO

stpcpy

EXAMPLES

```
#include <stdio.h>
#include <string.h>
```

Concatenate strings **Class: ANSI**

```
main()
{
char a[256],b[256];
int n;

while(1)
    {
    printf("\nEnter string A: ");
    if(gets(a) == NULL) exit(0);
    printf("Enter string B: ");
    if(gets(b) == NULL) exit(0);
    printf("Enter maximum length N: ");
    scanf("%d",&n);
    printf("strcat(A,B):    \"%s\"\n",strcat(a,b));
    printf("strncat(A,B,N): \"%s\"\n",strncat(a,b,n));
    }
```

Compare strings **Class: ANSI**

NAME

```
strcmp.............Compare strings
strcmpi............Compare strings, case-insensitive
stricmp............Compare strings, case-insensitive
strncmp............Compare strings, length-limited
strnicmp...........Compare strings, no case, max size
```

SYNOPSIS

```
#include <string.h>

x = strcmp(a,b);
x = strcmpi(a,b);
x = stricmp(a,b);
x = strncmp(a,b,n);
x = strnicmp(a,b,n);

int x;          comparison result
char *a,*b;     strings being compared
```

DESCRIPTION

These functions compare two null-terminated strings. The ASCII collating sequence is used in all cases, but **strcmpi**, **stricmp** and **strnicmp** do not distinguish between upper and lower case. Note that **strcmpi** is exactly the same as **stricmp**, except that the former is a hold-over from various Microsoft compilers, while the latter is in the ANSI standard.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

```
NEGATIVE => first string is below second
ZERO     => strings are equal
POSITIVE => first string is above second
```

If the strings have different lengths, the shorter one is treated as if it were extended with zeroes. For **strncmp** and **strnicmp**, no more than **n** characters are compared.

RETURNS

As noted above.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char *p,a[256],b[256];
int n,x;

while(1)
   {
   printf("Enter string A: ");
   if(gets(a) == NULL) exit(0);
   printf("Enter string B: ");
   if(gets(b) == NULL) exit(0);
   printf("Enter maximum compare length: ");
   scanf("%d",&n);

   result("strcmp:  ",strcmp(a,b));
   result("stricmp: ",stricmp(a,b));
   result("strncmp: ",strncmp(a,b,n));
   result("strnicmp:",strnicmp(a,b,n));
   }
}


void result(name,r)
```

Compare strings **Class: ANSI**

```
char *name;
int r;
{
char *p;

if(x == 0) p = "is equal to";
if(x < 0) p = "is less than";
if(x > 0) p = "is greater than";
printf("%s String A %s string B\n",name,p);
}
```

strdup

Duplicate a string **Class: XENIX**

NAME

strdup..............Duplicate a string

SYNOPSIS

```
#include <string.h>

p = strdup(s);

char *p;        points to duplicate string
char *s;        points to string being duplicated
```

DESCRIPTION

This function creates a duplicate of the specified string by using **malloc** and **strcpy** to allocate space and copy the string to it.

RETURNS

A null pointer is returned if **malloc** fails. Otherwise, the function returns a pointer to the duplicate string.

Insert a string **Class: LATTICE**

NAME
strins..............Insert a string

SYNOPSIS
```
#include <string.h>

void strins(to,from);

char *to;        destination string
char *from;      source string
```

DESCRIPTION
This function inserts the source string in front of the destination. Both strings must be null-terminated, and the destination is shifted to the right (upward in memory) in order to accomodate the source string. The final result is a single null-terminated string.

CAUTIONS
Make sure that the destination area is large enough.

SEE ALSO
strcat

EXAMPLES
```
#include <string.h>
char here[] = "Here ";
char now[30] = "and now";
printf("%s, %s\n",here,now);
strins(now,here);        /* now => "Here and now" */
printf("%s\n",now);
```

strlen,stclen

Measure length of a string **Class: UNIX**

NAME

stclen..............Measure length of a string
strlen..............Measure length of a string

SYNOPSIS

```
length = strlen(s);
length = stclen(s);
int length;             number of bytes in "s" (before null)
```

DESCRIPTION

These functions return the number of bytes in string **s** before the null terminator byte.

RETURNS

length = number of bytes in string before null byte

EXAMPLES

```
x = strlen("abc");        /* x is 3 */
x = strlen("");           /* x is 0 */
```

Change case of string **Class: XENIX**

NAME

strlwr..............Convert string to lower case
strupr..............Convert string to upper case

SYNOPSIS

```
#include <string.h>

p = strlwr(s);
p = strupr(s);

char *p;        return pointer (same as s)
char *s;        string pointer
```

DESCRIPTION

These functions convert all alphabetic characters in the specified null-terminated string to lower or upper case. In each case, the function return value is the same as the string pointer.

RETURNS

Both functions return the original string pointer.

strmfe

Make file name with extension **Class: LATTICE**

NAME

strmfe..............Make file name with extension

SYNOPSIS

```
#include <string.h>

strmfe(newname,oldname,ext);

char *newname;  new file name
char *oldname;  old file name
char *ext;      extension
```

DESCRIPTION

This function copies the old file name to the new name, deleting any extension. Then it appends the specified extension to the new file name, with an intervening period. For example,

OLDNAME	EXT	NEWNAME
"c:myprog.c"	"cc"	"c:myprog.cc"
"abc"	"exe"	"abc.exe"

CAUTIONS

The **newname** area must be large enough to accept the file name string and the separator. A safe size is **FMSIZE**, which is defined in the **dos.h** header file.

SEE ALSO

strmfn, strmfp

Make file name from components **Class: LATTICE**

NAME

strmfn..............Make file name from components

SYNOPSIS

```
#include <string.h>

strmfn(file,drive,path,node,ext);

char *file;       file name pointer
char *drive;      drive code pointer
char *path;       directory path pointer
char *node;       node pointer
char *ext;        extension pointer
```

DESCRIPTION

This function makes a file name from four possible components. In general, the name is constructed as follows:

```
drive:path/node.ext
```

If the **drive** pointer is not null, that string is moved to the area pointed to by the **file** argument. Then a colon is inserted unless one is already there. Next, if **path** is not NULL, it is appended to **file,** and the directory separator specified by _SLASH is added if necessary. The **node** string is appended next, unless it is NULL. Finally, if **ext** is not NULL, a period is appended to **file,** followed by the **ext** string.

RETURNS

None

Make file name from components **Class: LATTICE**

CAUTIONS

Make sure that the **file** pointer refers to an area that is large enough to hold the result. A safe value is **FMSIZE,** which is defined in **dos.h.**

SEE ALSO

strmfe, strmfp, _SLASH

EXAMPLES

```
#include <dos.h>
#include <stdio.h>
#include <string.h>

char buffer[FMSIZE];

/* The next statements both place "abc/def/ghi"
   into the buffer.  */


strmfn(buffer,NULL,"abc/def","ghi",NULL);
strmfn(buffer,NULL,"abc/def/","ghi",NULL);


/* The next statements both generate
   "df0:myfile.str"  */

strmfn(buffer,"df0",NULL,"myfile","str");
strmfn(buffer,"df0:",NULL,"myfile","str");
```

Make file name from path/node **Class: LATTICE**

NAME

strmfp..............Make file name from path/node

SYNOPSIS

```
#include <string.h>

strmfp(name,path,node);

char *name;      file name
char *path;      directory path
char *node;      node
```

DESCRIPTION

This function copies the path string to the file name area, appending the **_SLASH** separator if the path string is not empty and does not end with a slash, backslash, or colon. Then the node string is appended to the file name. **_SLASH** is an external character variable that defaults to a slash (/).

CAUTIONS

The **name** area must be large enough to accept the file name string. A safe value is **FMSIZE,** which is defined in the **dos.h** header file.

SEE ALSO

strmfe, strmfn, _SLASH

strrev

Reverse a character string **Class: XENIX**

NAME

strrev..............Reverse a character string

SYNOPSIS

```
#include <string.h>

p = strrev(s);

char *p,*s;     string pointer
```

DESCRIPTION

This function reverses a character string. That is, it "rotates" the string about its mid-point such that the last character is first and the first is last.

RETURNS

Returns same pointer that was passed to it.

Set string to value **Class: XENIX**

NAME

strnset.............Set string to value, max length
strset..............Set string to value

SYNOPSIS

```
#include <string.h>

p = strnset(s,c,n);
p = strset(s,c);

char *p;          return pointer (same as s)
char *s;          string pointer
int c;            value
int n;            maximum string length
```

DESCRIPTION

These functions set all bytes of a null-terminated string to the same value, not including the terminator byte. For **strnset** no more than **n** bytes will be set. That is, if a null byte is not found by the time **n** bytes have been set, the operation stops.

RETURNS

Both functions return the original string pointer.

strsfn

Split file name **Class: LATTICE**

NAME

strsfn..............Split file name

SYNOPSIS

```
#include <string.h>

strsfn(file,drive,path,node,ext);

char *file;       file name pointer
char *drive;      drive code pointer
char *path;       directory path pointer
char *node;       node pointer
char *ext;        extension pointer
```

DESCRIPTION

This function splits a file name into four possible components and places them into the **drive, path, node,** and **ext** strings. If any of those arguments are NULL, then those components are discarded.

In general, a complete file name is constructed as follows:

```
drive:path/node.ext
```

When **strsfn** splits the file name, it leaves the colon attached to the drive code, but drops the other punctuation characters. In other words, the **path** component will not end with a slash, and the **ext** component will not begin with a period. Slashes or backslashes within the **path** component are preserved.

RETURNS

None

SEE ALSO

strgfn, strmfe, strmfn

EXAMPLES

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

char a[3],b[FMSIZE],c[FNSIZE],d[4];

/* After the next statement, the component strings are:

        a => ""
        b => "abc/def"
        c => "ghi"
        d => ""

*/

strsfn("abc/def/ghi",a,b,c,d);


/* After the next statement, the component strings are:

        a => "b:"
        b => ""
        c => "myfile"
        d => "str"

*/

strsfn("b:myfile.str",a,b,c,d);
```

CAUTIONS

You must make sure that the **drive, path, node,** and **ext**

pointer refer to areas that are large enough to hold the largest string that might be generated. Note that **strsfn** does not check that any component lengths are exceeded, although it does copy **file** string to an internal buffer of size FMSIZE and truncate it if it is too long. If you want to be absolutely sure that no overflows occur, make each component area be FMSIZE bytes long.

EXAMPLES

```
/*
*
* This program splits file names that it reads
* from stdin.
*
*/
#include <stdio.h>
#include <string.h>

main()
{
char input[80],drive[80],path[80],node[80],ext[80];

while(1)
        {
        puts("\nEnter file name...\n");
        if(gets(input) == NULL) exit(0);
        strsfn(input,drive,path,node,ext);
        printf("DRIVE: \"%s\"\n",drive);
        printf("PATH:  \"%s\"\n",path);
        printf("NODE:  \"%s\"\n",node);
        printf("EXT:   \"%s\"\n",ext);
        }
}
```

NAME

strsrt..............Sort string pointer list

SYNOPSIS

```
#include <string.h>

strsrt(s,n);

char *s[];         string pointer list
int n;             number of pointers in list
```

DESCRIPTION

This function performs a simple bubble sort of the string pointers in the specified list. It is particularly useful in conjunction with the **getfnl** and **strbpl** functions. For large lists, you will usually get better performance using **tqsort**.

RETURNS

None.

SEE ALSO

getfnl, strbpl, tqsort

EXAMPLES

```
/*
* This program constructs an array of pointers
* to all file names in the current directory
* that have an extension of ".c".  Then the
* array is sorted into ASCII order.
*
*/
```

Sort string pointer list **Class: LATTICE**

```
char names[3000],*pointers[300];
int count;

count = getfnl("*.c",names,sizeof(names),0);
if(count > 0)
   {
   if(strbpl(pointers,300,names) != count) goto error;
   strsrt(pointers,count);
   }
```

Get a token **Class: ANSI**

NAME

strtok.............Get a token

SYNOPSIS

```
#include <string.h>

t = strtok(s,b);

char *t;          token pointer
char *s;          input string pointer or NULL
char *b;          break character string pointer
```

DESCRIPTION

This function treats the input string as a series of one or more tokens separated by one or more characters from the break string. By making a sequence of calls to **strtok**, you can obtain the tokens in left-to-right order. To get the first (leftmost) token, supply a non-NULL pointer for the **s** argument. Then to get the next tokens, call the function repeatedly with a NULL pointer for **s**, until you get a NULL return pointer to indicate that there are no more tokens. The break string can be changed from one call to another.

Each time it is entered, **strtok** takes the following steps:

1. If the input string is NULL, obtain the string pointer that was used on the preceding call. Otherwise use the new input string pointer.

2. Scan forward through the string to the next non-break character. If it is a null byte, return a value of NULL to indicate that there are no more tokens.

3. Scan forward through the string to the next break character or the null terminator. In the former case, write a null byte into the string to terminate the token, and then scan forward until the next non-break is found. In either case, save the final value of the string pointer for the next call, and return the token pointer.

Note that the input string gets changed as the scan progresses. Specifically, a null byte is written at the end of each token.

RETURNS

A NULL pointer is returned when there are no more tokens.

SEE ALSO

stptok, strcspn, strspn

EXAMPLES

```
/*
* This example breaks out words that are separated
* by blanks or commas. The token pointer takes on
* the following values as the program loops:
*
*        LOOP    TOKEN
*          1     "first"
*          2     "second"
*          3     "third"
*          4     "fourth"
*          5     NULL
*
*/
#include <string.h>
```

Get a token **Class: ANSI**

```
#include <stdio.h>

char test[] = "first, second third, fourth";

char *token;
token = strtok(test,", ");
while(token != NULL)
  {
  printf("%s\n",token);
  token = strtok(NULL,", ");
  }
}
```

strtol

Convert string to long integer **Class: UNIX**

NAME

strtol.............Convert string to long integer

SYNOPSIS

```
#include <string.h>

r = strtol(p,np,base);

long r;          result
char *p;         input string pointer
char **np;       receives new input string pointer
int base;        conversion base
```

DESCRIPTION

This function converts an ASCII input string into a long integer according to the specified base, which can range from 0 to 36, excluding 1. Valid digit characters are 0 to 9, a to z, and A to Z. The highest allowable character is determined by the conversion base. For example, if the base is 17, then the string can contain digits from 0 to 9, a to g, and A to G.

The function skips leading white space and then checks for a leading plus or minus sign. In the latter case, the result of the conversion is negated before it is returned. The conversion stops at the first invalid character, and a pointer to that character is returned in **np** if the **np** argument is not NULL. Note that if the entire string is converted, **np** will contain a pointer to the null terminator byte.

If **base** is 0, then the string is analyzed to see if it is octal, decimal, or hexadecimal, as follows:

Convert string to long integer **Class: UNIX**

Base 16 If the string begins with 0x or 0X, base 16 (hexadecimal) conversion is performed.

Base 8 Otherwise, if the string begins with 0, base 8 (octal) conversion is performed.

Base 10 If neither of the above applies, base 10 (decimal) conversion is performed.

CAUTIONS

No check is made for overflow.

SEE ALSO

atol, stcd_l

EXAMPLES

```
/*
*
* This program tests the strtol function.
*
*/
#include <stdio.h>
#include <string.h>

main()
{
char *p,buff[80];
int base;
long x;

while(1)
        {
        printf("\nEnter number base (0 to 36): ");
```

```
            if(gets(buff) == NULL) exit(0);
            if(buff[0] == '\0') exit(0);
            base = atoi(buff);
            if((base < 0) || (base > 36)) continue;

            printf("Enter number: ");
            if(gets(buff) == NULL) exit(0);
            if(buff[0] == '\0') exit(0);
            x = strtol(buff,&p,base);
            printf("Decimal result = %ld\n",x);
            if(*p != '\0') printf("Residual = %s\n",p);
            }
    }
```

NAME

stspfp..............Parse file path

SYNOPSIS

```
#include <string.h>

error = stspfp(path,nx);

int error;        -1 for error, 0 for success
char *path;       file path string
int nx[16];       node index array
```

DESCRIPTION

This function parses a file path, which is a null-terminated string consisting of nodes separated by the **_SLASH** character. Each separator is replaced with a null byte, and the index to the first character of that node is placed into the node index array. The last entry in the array is followed by a -1. A leading separator in the path string is skipped.

RETURNS

A return value of -1 indicates that the path contains more than 15 nodes.

SEE ALSO

stcgfe, stcgfn, stcgfp, strsfn

EXAMPLES

```
/*
*
* The following piece of code parses /ABC/DE/F
```

```
* into strings ABC, DE, and F. The node index
* array will then contain 1, 5, 8, and -1.
*
*/
int xx[16];

stspfp("/ABC/DE/F",xx);
```

Call system command processor **Class: ANSI**

NAME

system..............Call system command processor

SYNOPSIS

```
#include <stdlib.h>

error = system(cmd);

int error;        non-zero if error
char *cmd;        command string
```

DESCRIPTION

This function invokes the system command processor and passes the **cmd** string to it. Under AmigaDOS, this function invokes the AmigaDOS **Execute** facility.

RETURNS

If the command processor cannot be invoked, a value of -1 is returned, and additional error information can be found in **errno** and **_OSERR**. Otherwise, the function returns the value passed back by the command processor.

SEE ALSO

errno, _OSERR

EXAMPLES

```
#include <stdlib.h>
int x;
x = system("copy #? to df1:");
if(x < 0) printf("System command failed\n");
if(x > 0) printf("System command error %d\n",x);
```

time

Get system time in seconds **Class: ANSI**

NAME

time................Get system time in seconds

SYNOPSIS

```
#include <time.h>

timeval = time(timeptr);

long timeval;              time value
long *timeptr;             pointer to time value storage
```

DESCRIPTION

This function returns the current time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. If **timeptr** is not NULL, the time value is also stored in that location.

SEE ALSO

asctime, ctime, ftime, gmtime, localtime, tzset, utime

EXAMPLES

```
#include <time.h>
#include <stdio.h>

main()
{
long t;

time(&t);
printf("Current time is %s\n",ctime(&t));
}
```

Character conversion functions **Class: ANSI**

NAME

toascii............Convert character to ASCII
tolower............Convert character to lower case
toupper............Convert character to upper case

SYNOPSIS

```
#include <ctype.h>

cc = toascii(c);
cc = tolower(c);
cc = toupper(c);

int cc;         converted character
int c;          character to convert
```

DESCRIPTION

These functions convert characters into different forms. If you include **ctype.h** as shown above, they are actually defined as macros and produce in-line code to perform the conversions. Without **ctype.h,** they are actual functions resolved in the standard library. If you want to use the function version but must include **ctype.h** for some other reason, use **#undef** to undefine the conversion macros.

The **toascii** conversion simply resets all high-order bits, leaving only the lower seven. The **tolower** conversion tests if **c** is an upper case alphabetic character and, if so, converts it to lower case. Otherwise, **cc** is the same as **c**. The **toupper** conversion is the reverse of **tolower**.

SEE ALSO

_ctype

EXAMPLES

```
/*
*
* The following program echoes each input
* line in upper case.
*
*/
#include <stdio.h>
#include <ctype.h>

main()
{
char b[100],*p;

while(gets(b) != NULL)
        {
        for(p = b; *p != '\0'; p++) *p = toupper(*p);
        puts(b);
        }
}
```

Trigonometric functions **Class: UNIX**

NAME

```
cos..................Cosine function
sin..................Sine function
tan..................Tangent function

acos.................Arccosine function
asin.................Arcsine function
atan.................Arctangent function
atan2................Arctangent of x/y

cosh.................Hyperbolic cosine function
sinh.................Hyperbolic sine function
tanh.................Hyperbolic tangent function
```

SYNOPSIS

```
#include <math.h>

r = cos(x);
r = sin(x);
r = tan(x);

r = acos(x);
r = asin(x);
r = atan(x);
r = atan2(x,y);

r = cosh(x);
r = sinh(x);
r = tanh(x);

double r;        result;
double x,y;      arguments
```

DESCRIPTION

The **cos**, **sin**, and **tan** routines compute the normal

circular functions of angles expressed in radians.

The **acos, asin, atan** and **atan2** routines compute the inverse circular functions, returning angular values expressed in radians. Results are constrained as follows:

FUNCTION	RETURN RANGE
acos	0 to PI
asin	-PI/2 to PI/2
atan	-PI/2 to PI/2
atan2	-PI/2 to PI/2

Since the tangent becomes very large for angles close to PI/2, the **atan2** function is often used to avoid computations with large numbers that might easily overflow. With **atan2,** you can express the large tangent value as a quotient of two more reasonable numbers.

The **cosh, sinh,** and **tanh** routines compute the normal hyperbolic functions.

SEE ALSO

matherr

Set time zone variables **Class: XENIX**

NAME

tzset..............Set time zone variables

SYNOPSIS

```
#include <time.h>

tzset();

/*    These symbols are defined in time.h:
 *
 *        extern int daylight;
 *        extern long timezone;
 *        extern char *tzname[2];
 *        extern char tzstn[4];
 *        extern char tzdtn[4];
 *
 */
```

DESCRIPTION

The **tzset** function assigns values to the time zone variables **daylight**, **timezone**, and **tzname**. These variables are then used by **localtime** and other functions to correct from Greenwich Mean Time (GMT) to local time.

The values for these variables are obtained from the character string pointer named **_TZ**, which has the form:

```
    char *_TZ = "aaabbbccc"
```

where **aaa** is the 3-letter abbreviation for the local standard time zone (e.g. CST), and **bbb** is a number from -23 to +24 indicating the value that is subtracted from GMT in order to obtain local standard

Set time zone variables **Class: XENIX**

time. Both **aaa** and **bbb** are required, but **ccc** is the abbreviation for the local daylight savings time zone (e.g. CDT), and it should be present only if daylight savings time is currently in effect. Initially, **_TZ** is set to NULL. It should be initialized with the address of a string corresponding to the correct time zone. If **_TZ** is NULL, **tzset** uses the default string "CDT6".

When **tzset** is called, first **timezone** is loaded with the number of seconds that must be subtracted from GMT in orderd to get the local time. Next **daylight** is loaded with 0 if the **ccc** portion of **_TZ** is absent and 1 if **ccc** is present. Then the **aaa** and **ccc** parts are copied to **tzstn** and **tzdtn,** respectively, with null terminators. Finally, **tzname[0]** and **tzname[1]** are loaded with pointers to **tzstn** and **tzdtn,** respectively.

RETURNS

None.

CAUTIONS

The **tzstn** and **tzdtn** variables are Lattice extensions and will not exist on the typical UNIX system.

SEE ALSO

timedata, localtime

Push input character back **Class: ANSI**

NAME

ungetc..............Push input character back

SYNOPSIS

```
r = ungetc(c, fp);
int r;              return character or code
char c;             character to be pushed back
FILE *fp;           file pointer
```

DESCRIPTION

This function pushes a character back to the specified Level 2 input file. The character need not be the same as the one that was most recently read. However, before calling **ungetc**, you must have read at least one character via **fgetc** or one of the other Level 2 input functions. Also, you can only push back one character; if you call **ungetc** more than once between input functions, the results are undefined.

RETURNS

Normally **ungetc** returns the character that was pushed back. However, if the end-of-file has been hit or if no characters have been read yet, the value EOF is returned.

EXAMPLES

```
#include <stdio.h>

main()
{
int c;

while(1)
```

```
      {
      printf("Loop 1...\n");
      while((c = getchar()) != EOF)
        {
        if(isalpha(c)) putchar(c);
        else break;
        }
      ungetc(c);
      printf("\n\nLoop 2...\n");
      while((c = getchar()) != EOF)
        {
        if(isalpha(c) == 0) putchar(c);
        else break;
        }
      ungetc(c);
      }
    printf("\n\nDone\n");
    }
```

Pack or unpack UNIX time **Class: LATTICE**

NAME

utpack..............Pack UNIX time
utunpk..............Unpack UNIX time

SYNOPSIS

```
#include <stdlib.h>

ut = utpack(x);
utunpk(ut,x);

long ut;        packed UNIX time
char *x;        unpacked UNIX time
```

DESCRIPTION

These functions pack and unpack the 32-bit value time that is traditionally used in UNIX systems. This value is the number of seconds since 00:00:00, January 1, 1970. The **time** function returns the system clock in this form relative to Greenwich Mean Time.

The unpacked time is a 6-byte array in the following format:

```
x[0] => year - 1970 (-128 to +127)
x[1] => month (1 to 12)
x[2] => day (1 to 31)
x[3] => hour (0 to 23)
x[4] => minute (0 to 59)
x[5] => second (0 to 59)
```

Although this array is similar to the one produced by **getclk** and used by **stpdate**, note that the year is biased relative to 1970 instead of 1980. So, if you use **utunpk** followed by **stpdate**, you must subtract 10 from **x[0]** before the **stpdate** call. Note also that the

year is a signed character and can be negative. A value of -3, for example, is 1967 (i.e. 1970 - 3).

SEE ALSO

ctime, getclk, gmtime, localtime, stpdate, time

EXAMPLES

```
/*
*
* Get a file time and convert it to UNIX time.
* No error checks.
*
*/
#include <time.h>
#include <dos.h>
#include <stdlib.h>
main(argc,argv)
int argc;
char *argv[];
{
char tt[6];
int fh;
long ft,ut;

ft = getft(argv[1]);
tt[0] -= 10;
ut = utpack(tt);
printf("File time is: %s\n",ctime(&ut));
}
```

SECTION X
COMBINED INDEX

This section inicludes all commands (Section C), data names (Section D), environment variables (Section E) and function names (Section F). A local index is also provided at the beginning of each section.

Become A Lattice® Registered User

Join the Lattice Family as a Registered User and receive the full benefits of *Lattice Service*:

- Your *Lattice Update Card*™ which entitles you to receive low-priced product updates. We cannot update your product unless you send us your *Lattice Update Card*.
- A subscription to our *Lattice Works*™ newsletter.
- Access to the *Lattice Bulletin Board Service*.
- Access to the *Lattice Technical Support Hotline*.
- Advance notification of new versions and new products.
- Special offers made only to Lattice Registered Users.

Protect your investment. Complete the postage paid card below and mail it today!

The number pre-stamped on this card is your product serial number. You will need this number when contacting our *Technical Support Hotline*.

Serial Number: AM01- 62992

The card below is your *Lattice Customer Registration Card*. It is for your protection and is informational only. This is *not* a warranty registration. If you have more than one product, please register each product using a separate card.

*****READ CAREFULLY BEFORE OPENING*****

BY OPENING THIS PACKAGE YOU INDICATE YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS.

PROGRAM LICENSE AND DISTRIBUTION AGREEMENT

1. This software product is licensed to you for your personal use on a single computer system at a time. You must return the enclosed registration card to Lattice in order to identify yourself as the Licensee.

2. You may make backup copies of the program diskette(s) and may also copy the software to another storage medium for your personal use. You must remove all copies from the machine that you are no longer using.

3. You may not give copies of the product to another person, transfer the license to another person, or share the use of the product with others (e.g. via a network) without a specific agreement with Lattice.

4. Lattice reserves the right to terminate this license upon breach. You may terminate the license at any time by destroying all copies of the product. You should also notify Lattice at that time so that you will be removed from the software update notification list.

5. You may distribute derivative works (i.e. programs generated by and/or including binary code parts of the Lattice product in such a way that they cannot be separated out) as you desire with no further fees paid to Lattice. Source code of the product may NOT be distributed without a specific agreement with Lattice.

6. Product specifications and features are subject to change without notice.

7. This agreement may be superseded by a direct agreement with Lattice.

8. The enclosed Master Diskette(s) are your proof of purchase and must be returned to Lattice in order to receive product updates or upgrades when they are made available. However, if Lattice has not received your registration card, Lattice is under no obligation to make availiable to you any updates or upgrades even though you may have made payments of any applicable fee.

LIMITED WARRANTY

1. Lattice warrants the diskette(s) on which the program is furnished to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your sales receipt.

2. Lattice does NOT warrant that the program will meet your requirements or that the operation of the program will be uninterrupted and error free. You are solely responsible for the selection of the program to achieve your intended results and for the results actually obtained.

THIS AGREEMENT WILL BE GOVERNED BY THE LAWS OF THE STATE OF ILLINOIS

Lattice AmigaDOS
C-Compiler
Version 3.10
Serial AM01
Format AMIGA 880K

Lattice

Lattice, Incorporated

Lattice

Lattice, Incorporated
Post Office Box 3072
Glen Ellyn, Illinois 60138
TWX 910-291-2190
Telex 532253